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 $\Rightarrow$ error-prone

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

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
**Grow-only Set, state-based**

**Payload** = set \( A \)

- \( \text{add (atom } a) \)
- \( \text{merge (x,y) = x } \cup \text{ y} \)

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

- **Assume: state eventually delivered**
- **Why not remove()?**
- **Trial and error...**
- **Hmm, let's move on to something else**
Counter, operation-based

Operations
- $\text{increment} ()$
- $\text{decrement} ()$
- $\text{read} () : \text{integer}$

Payload: $\text{val} = \text{integer}$

$\text{increment} (), \text{decrement} ()$
- empty precondition
- increment, decrement local

Transmit operations

• Let's look at the operations
  • simple, obviously correct spec
  • assume ops eventually delivered

• Easy: addition and subtraction commute
Counter, state-based

Increment-only:
- Payload: \( v = [\text{int}, \text{int}, \ldots] \)
- \( \text{value()} = \sum_i x[i] \)
- \( \text{increment()} = v[\text{MyID}]++ \)
- \( \text{merge}(x,y) = x \sqcup y = [\ldots, \max(x.vv[i],y.v[i]),\ldots] \)

Increment/decrement
- increment-only counters \( P, M \)
- \( \text{value()} = |P| - |M| \)
- \( \text{increment()}: \) add to \( P \)
- \( \text{decrement()}: \) add to \( M \)
- Positive or negative

A principled approach to eventual consistency
State-based replication

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

Update at source $x_1.f(u), x_2.g(), \ldots$
• 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
Semi-lattice

A poset \((S, \leq)\) is a *join-semilattice* if:

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

**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:**

- \((\text{int}, \leq)\) \(x \sqcup y = \max(x, y)\)
- \((\text{sets}, \subseteq)\) \(x \sqcup y = x \cup y\)
State-based convergent objects: CvRDT

If

• payload type forms a semi-lattice
• updates are monotonically increasing
• \( \text{merge} \) computes \( \sqcup \)

then replicas converge to LUB of last values

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

Example CvRDT

If

- payload type forms a semi-lattice
- updates are non-decreasing
- \text{merge} computes $\sqcup$

then replicas converge to LUB of last values

Example: $f = \text{assign}$, $\text{merge} = \text{max}$
Operation-based replication

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

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

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

Commutative-operation-based objects: CmRDTs

If:
• (Liveness) all replicas execute all operations in delivery 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
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
payload $X \ x$, timestamp $t$
  initial ⊥, 0
update assign $(X \ y)$
  $x, t := y, \text{now}()$
query value () : $X \ y$
  let $y = x$
compare $(R, S)$ : boolean $b$
  let $b = (R.t \leq S.t)$
merge $(R, S)$ : payload $T$
  if $R.t \leq S.t$ then $T.x, T.t = S.x, S.t$
  else $T.x, T.t = R.x, R.t$
A principled approach to eventual consistency

\[ \text{MV-Register} \]

\[ \approx \text{LWW-Set Register} \]
- **Payload**: \{ \((\text{value, VT vv})\) \}
- **assign**: overwrite \text{value}, \(\text{vv}++\)

Concurrent updates unioned (no lost updates)

\[ \text{merge } (X, Y) = \]
\[ \{ x \in X \mid \nexists y \in Y: x.\text{vv} < y.\text{vv} \} \cup \]
\[ \{ y \in Y \mid \nexists x \in X: x.\text{vv} > y.\text{vv} \} \]
“An add operation is never lost. However, deleted items can resurface.” [Dynamo, SOSP 2007]

Preferred approach: to design a proper Set CRDT
Operations:
• \textit{add} (atom \ a)
• \textit{remove} (atom \ a)
• \textit{lookup} (atom \ a) : boolean

No duplicates

The prototypical CRDT?
• \textit{remove} does not commute with \textit{add}
• Approximations: modify semantics

Set
Add-remove-add?

Doesn't converge: not a CRDT
  • ∪, \ don't commute
  • No obvious order x ≤ y
Solution: alter semantics
Add, remove: 2P-set

- **Payload** = \((\text{Grow-Set } A, \text{Grow-Set } R)\)
- **add** (atom \(a\))
  - **remove** (atom \(a\)) [ spre: \(a \in A\) ]
- **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** (\(x, y\)) = \((x.A \cup y.A, x.R \cup y.R)\)

- \(A=\) added
- \(R=\) removed (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
- 2P-Set indistinguishable from Set
- Correct shopping cart
Observed-Remove Set (state)

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

- Can never remove more tokens than exist
- Op order ⇒ removed tokens have been previously added

- Better shopping cart
- What anomalies?
OR-Set specification (op)

payload set $S$
   initial $\emptyset$

query $\text{lookup} \ (\text{element } e) : \text{boolean } b$
   let $b = (\exists u : (e, u) \in S)$

update $\text{remove} \ (\text{element } e)$
   atSource $(e)$
      pre $\text{lookup}(e)$
      let $R = \{(e, u) | \exists u : (e, u) \in S\}$

   downstream $(R)$
      pre $\forall (e, u) \in R : \text{add}(e, u)$ has been delivered
      $S := S \setminus R$

update $\text{add} \ (\text{element } e)$
   atSource $(e)$
      let $u = \text{unique}()$

   downstream $(e, u)$
      $S := S \cup \{(e, u)\}$
Map

Set of (key, value) pairs

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

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

CRDT approximations

- 2P-Map
- PN-Map
- LWWW 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: \texttt{addEdge}(u,v) || \texttt{removeVertex}(u)
- Discard when all concurrent \texttt{addEdge} delivered
  - i.e. when \texttt{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

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 of elements of type $T$

- Co-operative edit buffer: sequence of atoms
- *add-at-location*, *remove*
Elements of type \((\text{atom } v, \text{LTS } ts)\)
- Explicit (total order) graph \(x < y < z\)

\text{add-after} (x, y):
- \text{dpre}: \text{add-after}(..., x) \Rightarrow \text{add-after} (x, ...)
- \text{Sequential}: \text{add-after} (x,y) \Rightarrow \text{add-after} (x,z)
  \Rightarrow y.ts < z.ts \land x < z < y
- \text{Concurrent}: \text{add-after} (x,y) \parallel \text{add-after} (x,z)
  \land y.lts < z.lts \Rightarrow x < z < y
Roh's RGA

Elements of type \((\text{atom } v, \text{LTS ts})\)

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

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

- 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: \(\text{add-after} (x,y) \parallel \text{add-after} (x,z)\)
  \[\land y.ts < z.ts \Rightarrow x < z < y\]
Assign each element a unique real number
  • position

Real numbers not appropriate
  • approximate by tree
Assign each element a unique real number
  • position

Real numbers not appropriate
  • approximate by tree
Layered Treedoc

- sparse 8⁶⁴-ary tree
- linear array
- binary tree

Edit: Binary tree
Concurrency: Sparse tree
Compact: Linear array

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
Portfolio of CRDTs

Register
- Last-Writer Wins
- Multi-Value

Set
- Grow-Only
- 2P
- PN
- LWW
- Observed-Remove

Map
- \( \approx \) Set + Register

Counter
- Unlimited
- Non-negative

Graph
- Arbitrary
- Monotonic poset

Sequence
- Edit sequence
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
A principled approach to eventual consistency