State-based CRDTs

Paulo Sérgio Almeida

Universidade do Minho

State-based Replication

Conflict-free Replicated Data Types

- Provide operations, like standard abstract datatypes
- Each datatype object replicated and accessed locally
  - Mutator operations update state
  - Query operations look at state and return result
- Information propagated to other replicas asynchronously
- Object highly available even under partitions

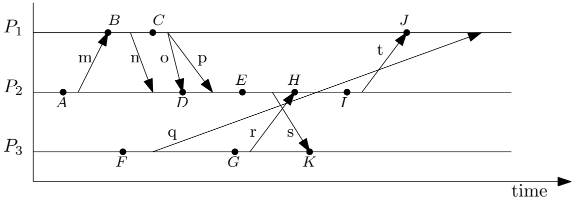
The C from CRDTs

- C. . . Replicated Data Types
  - Convergent?
  - Conflict-free?
  - Commutative?
- Convergence while resolving conflicts
- Replicas keep converging; world does not have to stop
- Conflicts are dealt with semantically: spec of datatype
- Availability is achieved by forgoing total orders
- Concurrent operations will become visible in different orders
- Some confusion about what is commutative (spec vs impl)
  - some operations are not (semantically) commutative
  - effects of executing concurrent operations must be

Operation-based vs state-based approaches

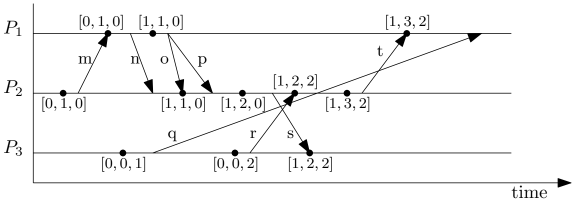
- State-based approaches
  - propagate replica states
  - detect mutual inconsistency
  - reconcile (merge) concurrent replicas
  - anti-entropy by opportunistic, "background" communication
  - can be made more incremental by delta-state approach
- Operation-based approaches
  - propagate information about operations
  - use a reliable messaging algorithm for propagation
  - need ordering guarantees (typically causal)
- Here we address state-based CRDTs

State-based replication through state propagation



- Messages carry state, immediately delivered
- In figure, marks signal state change, labelled by resulting state
- States $A \sqsubseteq B \sqsubseteq C \sqsubseteq D \sqsubseteq E$ evolve in sequence (even through different replicas)
- States E and G evolved concurrently, are conflicting ... ...and must be reconciled (merged) to obtain H
- $B = A$; also $A \sqsubseteq C$, so $D = C$; also $C \sqsubseteq I$, so $J = I$
- No state change when delivering n ($B = A$), p (duplicate message) and q ($F \sqsubseteq G \sqsubseteq H \sqsubseteq I = J$)

Detecting mutual inconsistency through version vectors



- Only updates increase self entry (not sends or receives)
- Send attaches *version vector* to message m as V_m
- Receive performs pointwise maximum $V_i := \max(V_i, V_m)$
  - assumes deterministic conflict resolution
  - otherwise, must also increase self entry
- VVs compared by standard comparison of functions

$$V_i \leq V_j \Leftrightarrow \forall p \cdot V_i[p] \leq V_j[p]$$

- Unlike with VCs, several nodes can have the same VV

How to resolve conflicts

- Version vectors know whether two replicas conflict
- If that happens, some conflict resolution must be performed
- Classically performed in some ad hoc way
- Conflict resolution (merge) should be
  - deterministic: result as a function of inputs
  - obviously a commutative function
  - associative: same result when merging in different orders
  - monotonic: merging with “newer” state produces “newer” state

Mathematical concept of *join semilattices* is the solution

State-based Replication

Lattices and Order

State-based CRDTs

Sets and partially ordered sets (posets)

- A set assumes no order between elements
- A partially ordered set (poset), is a set equipped with a binary relation $\sqsubseteq$ which is:
  - (reflexive) $p \sqsubseteq p$
  - (transitive) $o \sqsubseteq p \wedge p \sqsubseteq q \Rightarrow o \sqsubseteq q$
  - (anti-symmetric) $p \sqsubseteq q \wedge q \sqsubseteq p \Rightarrow p = q$
- Two unordered elements are called concurrent.
  - (concurrent) $p \parallel q \Leftrightarrow \neg(p \sqsubseteq q \vee q \sqsubseteq p)$
- Some posets have *bottom*, an element smaller than any other:

$$\forall p \in P \cdot \bot \sqsubseteq p$$

Join semilattices

- In poset P, an upper bound of subset S is some u such that

$$\forall s \in S \cdot s \sqsubseteq u$$

- The *least upper bound* exists, when there is an upper bound smaller than any other upper bound
- In that case it is called the *join* of S, denoted $\bigsqcup S$.
- For two elements, $\bigsqcup\{p, q\}$ is denoted by $p \sqcup q$
- A poset P is a join semilattice if there exists a least upper bound for any pair of elements p and q in P
- Properties of the join operator:
  - (idempotent) $p \sqcup p = p$
  - (commutative) $p \sqcup q = q \sqcup p$
  - (associative) $o \sqcup (p \sqcup q) = (o \sqcup p) \sqcup q$

Examples and non-examples of join semilattices

- Some join semilattices:
  - natural numbers $\mathbb{N}$; $\sqsubseteq\ =\ \leq$; $\sqcup = \max$; $\bot = 0$
  - booleans $\mathbb{B}$, with False $\sqsubset$ True; $\sqcup = \vee$; $\bot =$ False
  - any totally ordered set (*chain*)
- Some posets which are not join semilattices:
  - unordered posets (*antichain*)
    - all elements being incomparable
  - strings under prefix ordering (e.g., *small* $\sqsubseteq$ *smallest* $\parallel$ *smaller*)
    - all concurrent elements are not joinable

Lattice compositions (1)

- Cartesian product $A \times B$ of two join semilattices A and B

$$\begin{array}{rcl} (a_1, b_1) \sqsubseteq (a_2, b_2) & \Leftrightarrow & a_1 \sqsubseteq a_2 \wedge b_1 \sqsubseteq b_2 \\ (a_1, b_1) \sqcup (a_2, b_2) & = & (a_1 \sqcup a_2, b_1 \sqcup b_2) \end{array}$$

- Lexicographic product $A \boxtimes B$ of two join semilattices A and B, when B has a bottom or A is a chain

$$\begin{array}{rcl} (a_1, b_1) \sqsubseteq (a_2, b_2) & = & a_1 \sqsubset a_2 \vee (a_1 = a_2 \wedge b_1 \sqsubseteq b_2) \\ (a_1, b_1) \sqcup (a_2, b_2) & = & \begin{cases} (a_1, b_1) & \text{if } a_2 \sqsubset a_1 \\ (a_2, b_2) & \text{if } a_1 \sqsubset a_2 \\ (a_1, b_1 \sqcup b_2) & \text{if } a_1 = a_2 \\ (a_1 \sqcup a_2, \bot) & \text{if } a_1 \parallel a_2 \end{cases} \end{array}$$

Lattice compositions (2)

- Powerset $\mathcal{P}(S)$ for any set S

$$\sqsubseteq \; = \; \subseteq \qquad \sqcup \; = \; \cup$$

- Function space $A \to B$ from set A to join semilattice B

$$f \sqsubseteq g = \forall x \in A \cdot f(x) \sqsubseteq g(x) \qquad (f \sqcup g)(x) = f(x) \sqcup g(x)$$

- Maps (partial functions) $K \rightharpoonup V$, to join semilattice V with bottom, where missing keys implicitly yield bottom

$$m(k) = \begin{cases} v & \text{if } (k, v) \in m \\ \bot & \text{otherwise} \end{cases}$$

  are efficient representations of total function; same $\sqsubseteq$ and $\sqcup$

State-based Replication

Lattices and Order

State-based CRDTs

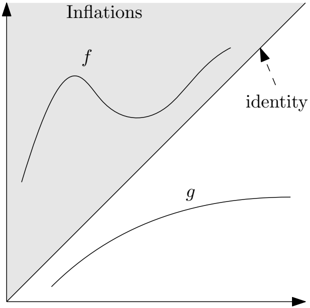
State-based CRDTs

- Operations applied locally
- State is a join-semilattice
- Replicas send the full state
- Replicas merge received state using join operator $\sqcup$
- Assume unreliable networks (loss, duplication, reordering)
- Sending full state ensures causal consistency

Update functions and state mutators

- Update functions invoke a state mutator
- State mutator must be an *inflation*
  - (inflation) $x \sqsubseteq f(x)$
  - (strict inflation) $x \sqsubset f(x)$
- What is needed is that state evolves monotonically
  - when updates are locally invoked
  - when remote state is received and merged through join

Inflations vs monotonic functions



- Erroneously, many places say mutators need to be monotonic
- State mutators need to be inflations
- Being monotonic is not necessary nor sufficient
- f is an inflation but not monotonic
- g is monotonic but not an inflation

Grow only set GSet$\langle E \rangle$

$$\begin{array}{rcl} \mathsf{GSet}\langle E \rangle & = & \mathcal{P}(E) \\ \bot & = & \{\} \\ \mathsf{add}_i(e, s) & = & s \cup \{e\} \\ \mathsf{elements}(s) & = & s \\ \mathsf{contains}(e, s) & = & e \in s \\ s \sqcup s' & = & s \cup s' \end{array}$$

- Trivial CRDT: state same as sequential datatype
- Add already an inflation; mutator = update
- Merging replica states by set union
- Anonymous CRDT: no need for node ids in the state

Single-writer principle and named CRDTs

- Powerful strategy: single writer principle
  - state partitioned in several parts
  - each node updates a part dedicated exclusively to itself
  - state is joined by joining respective parts
- Unique node ids can be used to partition state
  - as a map from ids to parts
  - example: version vectors
- CRDTs that use node ids in the state are *named CRDTs*

State-based PCounter

$$
\begin{aligned}
\mathsf{GCounter} &= \mathbb{I} \rightharpoonup \mathbb{N} \\
\bot &= \{\} \\
\mathsf{inc}_i(m) &= m\{i \mapsto m(i) + 1\} \\
\mathsf{value}(m) &= \sum_{j \in \mathbb{I}} m(j) \\
m \sqcup m' &= \{j \mapsto \max(m(j), m'(j)) \mid j \in \mathrm{dom}\, m \cup \mathrm{dom}\, m'\}
\end{aligned}
$$

- State maps replica ids to integers
- inc increments self entry (i)
- Merge is pointwise max
- Similar in structure to a version-vector

State-based PNCounter

$$\begin{aligned}
\mathsf{PNCounter} &= \mathsf{GCounter} \times \mathsf{GCounter} \\
\bot &= (\bot, \bot) \\
\mathsf{inc}_i((p, n)) &= (\mathsf{inc}_i(p), n) \\
\mathsf{dec}_i((p, n)) &= (p, \mathsf{inc}_i(n)) \\
\mathsf{value}((p, n)) &= \mathsf{value}(p) - \mathsf{value}(n) \\
(p, n) \sqcup (p', n') &= (p \sqcup p', n \sqcup n')
\end{aligned}$$

- Problem: dec is not natively an inflation
- Solved through a pair of GCounters (product composition)
  - increments and decrements tracked separately
  - counter value obtained as difference
- In practice, a single map to pairs is used

How to forget without tombstones

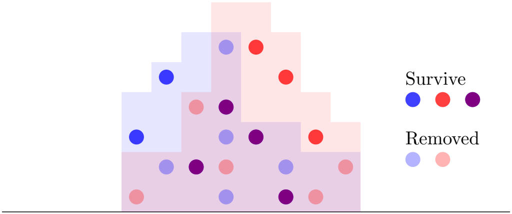
- Many times we want to remove things
  - example: set with add and remove
- But must always use inflations
  - cannot use single set for state and remove elements
- Using tombstones to mark removal makes state grow forever
- Example: two-phase set
  - allows add and remove
  - once removed, an element cannot be re-added
  - implementation with a pair of sets, to track adds and removes
  - state is always growing
- Tombstones can be avoided using *Causal CRDTs*

Causal CRDTs

- State has two components:
  - a *dot store*
  - a *causal context*
- The dot store (DS)
  - is a container for datatype-specific information
  - information tagged with unique event identifiers: *dots*
- The causal context (CC)
  - represents causal history: ids of all visible updates
  - in state-based CRDTs encoded by a version vector

> Any information covered by the causal context and not present in the dot store has already been removed

Joining causal CRDTs



- The dot store that results from a join of x and y
  - has dots from x's DS not covered by y's CC
  - has dots from y's DS not covered by x's CC
  - has the dots present in both DSs
- The causal context upon a join is the join of causal contexts

Some dot stores

$$
\begin{aligned}
& \mathsf{DS} \\
\mathsf{DotSet} : \mathsf{DS} &= \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\mathsf{DotFun}\langle V : \mathsf{Lattice}\rangle : \mathsf{DS} &= \mathbb{I} \times \mathbb{N} \rightharpoonup V \\
\mathsf{DotMap}\langle K,\ V : \mathsf{DS}\rangle : \mathsf{DS} &= K \rightharpoonup V
\end{aligned}
$$

- DotMap in particlar is a powerful concept
- Allows building map CRDTs which embed CRDTs, or maps which embed maps ... which embed CRDTs

$$\mathsf{DotMap}\langle K_1,\ \mathsf{DotMap}\langle K_2,\ \mathsf{DotFun}\langle \mathcal{P}(E)\rangle\rangle\rangle$$

Causal CRDTs

$$
\begin{aligned}
\mathsf{Causal}\langle T : \mathsf{DS}\rangle &= T \times \mathsf{CausalContext} \\
\sqcup &: \mathsf{Causal}\langle T\rangle \times \mathsf{Causal}\langle T\rangle \to \mathsf{Causal}\langle T\rangle \\
\\
\text{when} \quad & T : \mathsf{DotSet} \\
(s, c) \sqcup (s', c') &= ((s \cap s') \cup (s \setminus c') \cup (s' \setminus c), c \cup c') \\
\\
\text{when} \quad & T : \mathsf{DotFun}\langle \_ \rangle \\
(m, c) \sqcup (m', c') &= (\{k \mapsto m(k) \sqcup m'(k) \mid k \in \mathrm{dom}\ m \cap \mathrm{dom}\ m'\} \cup \\
& \quad \{(d, v) \in m \mid d \notin c'\} \cup \{(d, v) \in m' \mid d \notin c\}, c \cup c') \\
\\
\text{when} \quad & T : \mathsf{DotMap}\langle \_, \_ \rangle \\
(m, c) \sqcup (m', c') &= (\{k \mapsto \mathrm{v}(k) \mid k \in \mathrm{dom}\ m \cup \mathrm{dom}\ m' \wedge \mathrm{v}(k) \neq \bot\}, c \cup c') \\
& \text{where } \mathrm{v}(k) = \mathrm{fst}\,((m(k), c) \sqcup (m'(k), c'))
\end{aligned}
$$

Multi-value register MVReg$\langle E \rangle$

$$\begin{aligned}
\mathsf{MVReg}\langle E \rangle &= \mathsf{Causal}\langle \mathsf{DotFun}\langle E_{\perp}^{\top} \rangle\rangle \\
\mathsf{write}_i(v, (m, c)) &= (\{(i, c[i] + 1) \mapsto v\}, c\{i \mapsto c[i] + 1\}) \\
\mathsf{read}((m, c)) &= \{v \mid (k, v) \in m\}
\end{aligned}$$

- Dot store is a DotFun: map from dots to values in E
- Value is range of DotFun
- Previously defined join for the DotFun case used
  - value mapped from a given dot remains immutable
  - therefore, no join of values for any given key (dot) happens
- Join keeps concurrently written values

Observed-remove set $\mathsf{ORSet}\langle E \rangle$

$$
\begin{aligned}
\mathsf{ORSet}\langle E \rangle &= \mathsf{Causal}\langle \mathsf{DotMap}\langle E, \mathsf{DotSet} \rangle \rangle \\
\mathsf{add}_i(e, (m, c)) &= (m\{e \mapsto \{(i, c[i] + 1)\}\}, c\{i \mapsto c[i] + 1\}) \\
\mathsf{remove}_i(e, (m, c)) &= (\{e\} \lhd m, c) \\
\mathsf{elements}((m, c)) &= \mathsf{dom}\, m
\end{aligned}
$$

- Add replaces, for that key, any set of dots by a new singleton
- Remove simply removes that key (domain subtraction)
  - no new event introduced in the causal context
- Previously defined join for the DotMap case used
  - keeps concurrently added elements
  - removes elements removed elsewhere if no dot survives