Programming Distributed Systems

06 Replicated Data Types

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019
Motivation

So far, we resolved conflicting updates (i.e. non-commutative) updates simply by sequencing operations using arbitration order (ar).

But sometimes, applications

- do not want to depend on a global order such as \textit{ar}
- want to be made aware of conflicts
- want to resolve conflicts in a specific way
Example: Multi-value register

Standard Register (Last Writer Wins)

- **1**: \(\text{wr(“foo”)}: \text{ok}\)
- **2**: \(\text{wr(“bar”)}: \text{ok}\)
- **3**: \(\text{rd: “bar”}\)

Multi-Value Register (Amazon Dynamo)

- **1**: \(\text{wr(“foo”)}: \text{ok}\)
- **2**: \(\text{wr(“bar”)}: \text{ok}\)
- **3**: \(\text{rd: {“foo”, “bar”}}\)
How can we determine the state?

**Sequence-based conflict resolution**

1. \( \text{wr(“foo”)}: \text{ok} \)
2. \( \text{wr(“bar”)}: \text{ok} \)
3. \( \text{rd: “bar”} \)

\text{visible state} = \text{sequence of visible operations, sorted by arbitration order}

**GENERAL CONFLICT RESOLUTION**

\( \text{wr(“foo”)}: \text{ok} \)

\( \text{wr(“bar”)}: \text{ok} \)

\( \text{rd: {“foo”, “bar”}} \)

\text{visible state} = \text{subgraph of visible operations}
Formal model

- **Before:** \( S: \text{Op} \times \text{Op}^* \rightarrow \text{Val} \)
  - The operation we are performing
  - “Current state” = *Sequence of all Prior operations*
  - Returned Value

- **Now:** \( F: \text{Op} \times C \rightarrow \text{Val} \)
  - The operation we are performing
  - Operation Context = *Event Graph of visible operations*
  - Returned Value
Sequential semantics for registers

\[ S : Op \times Op^* \rightarrow Val \]

\[ S(rd(), wr(2) \cdot wr(8)) = 8 \text{ (read returns last value written)} \]

\[ S(rd(), \epsilon) = undef \]

\[ S(wr(3), rd() \cdot wr(2) \cdot wr(8)) = ok \text{ (write always returns ok)} \]
Operation Context

An operation context is a finite event graph $C = (E, \text{op}, \text{vis}, \text{ar})$.

- Events in $E$ capture what prior operations are visible to the operation that is to be performed.
- Models the situation at a single replica
Concurrent semantics for Multi-Value Register

\[ F : Op \times C \rightarrow Val \]

\[ F_{mvr}(wr(x), C) = \text{ok} \]

\[ F_{mvr}(rd(), C) = \{ x \mid \text{exists } e \text{ in } C \text{ such that } op(e) = wr(x) \text{ and } e \text{ is vis-maximal in } C \} \]
Quizz: What do the read ops return?
Return values in Abstract Executions revisited

- Previous lecture:

An abstract execution \((E, op, rval, rb, ss, vis, ar)\) satisfies a sequential semantics \(S\) if

\[
\text{rval}(e) = S(op(e), vis^{-1}.sort(ar))
\]

- Read-value consistency can also be defined wrt concurrency semantics

An abstract execution \(A = (E, op, rval, rb, ss, vis, ar)\) satisfies a concurrent semantics \(F\) if

\[
\text{rval}(e) = F(op(e), A |_{vis^{-1}(e), op, vis, ar})
\]
Conflict-free Replicated Data Types (CRDTs) [3]

- Same API as sequential abstract data type, but with concurrency semantics
- Catalogue of CRDTs
  - Register (Last-writer wins, Multi-value)
  - Set (Grow-Only, Add-Wins, Remove-Wins)
  - Flags
  - Counter (unlimited, restricted/bounded)
  - Graph (directed, monotone DAG)
  - Sequence / List
  - Map, JSON
- If operations are commutative, same semantics as in sequential execution
- Otherwise, need arbitration to resolve conflict
Specification: Replicated counter

- Operation \( inc \) commutes \( \Rightarrow \) No conflict resolution policy is needed
- Value returned depends only on \( E \) and \( op \), but not on \( vis \) and \( ar \)

\[
F_{ctr}(rd(), (E, op, vis, ar)) = |\{e' \in E | op(e') = inc\}|
\]
Semantics of a replicated Set or How to design a CRDT

- Sequential specification of abstract data type Set $S$:
  \[
  \begin{align*}
  \{\text{true}\} & \quad \text{add}(e) & \quad \{e \in S\} \\
  \{\text{true}\} & \quad \text{rmv}(e) & \quad \{e \notin S\}
  \end{align*}
  \]

- The following pairs of operations are commutative (for two elements $e, f$ and $e \neq f$):
  - $\{\text{true}\}$ add$(e)$; add$(e)$ $\{e \in S\}$
  - $\{\text{true}\}$ add$(e)$; add$(f)$ $\{e, f \in S\}$
  - $\{\text{true}\}$ rmv$(e)$; rmv$(e)$ $\{e \notin S\}$
  - $\{\text{true}\}$ rmv$(e)$; rmv$(f)$ $\{e, f \notin S\}$
  - $\{\text{true}\}$ add$(e)$; rmv$(f)$ $\{e \in S, f \notin S\}$

$\Rightarrow$ For these ops, the concurrent execution should yield the same result as executing the ops in any order.
What are the options regarding a concurrency semantics for add(e) and rmv(e)?

- The operations add(e) and rmv(e) are *not* commutative
  - \{true\} add(e); rmv(e) \{e \not\in S\}
  - \{true\} rmv(e); add(e) \{e \in S\}
- Options for conflict-resolution strategy when *concurrently* executing add(e) and rmv(e)
  - add-wins: \( e \in S \)
  - remove-wins: \( e \not\in S \)
  - erroneous state (i.e. escalate the conflict to the user)
  - last-writer wins (i.e. define arbitration order through total order, e.g., by adding totally-ordered timestamps)
Set Semantics

Standard Set

Add-Wins Set
Formal Semantics for the Add-Wins Set

\[
\begin{align*}
F_{\text{aws}}(\text{add}(x), C) &= \text{ok} \\
F_{\text{aws}}(\text{rmv}(x), C) &= \text{ok} \\
F_{\text{aws}}(\text{rd}(), C) &= \{ x \mid \text{exists } e \text{ in } C \text{ such that } \text{op}(e) = \text{add}(x) \\
&\quad \text{and there exists no } e' \text{ in } C \text{ such that } \\
&\quad \text{op}(e') = \text{rmv}(x) \text{ and } e \xrightarrow{\text{vis}} e' \}\n\end{align*}
\]
Sets with “interesting” semantics

- **Grow-only set**
  - Convergence by union on element set
  - No remove operation

- **2P-Set (Wuu & Bernstein PODC 1984)**
  - Set of added elements + set of tombstones (= removed elements)
  - Add/remove each element once
  - Problem: Violates sequential spec

- **c-set (Sovran et al., SOSP 2011)**
  - Count for each element how often it was added and removed
  - Problem: Violates sequential spec
Take a break!

A Mathematician, a Biologist and a Physicist are sitting in a street cafe watching people going in and coming out of the house on the other side of the street. First they see two people going into the house. Time passes. After a while they notice three persons coming out of the house.

The Physicist: “The measurement wasn’t accurate.”.

The Biologist: “They have reproduced”.

The Mathematician: “If now exactly one person enters the house then it will be empty again.”
CRDTs: Strong Eventual Consistency

- **Eventual delivery**: Every update is eventually applied at all correct replicas.
- **Termination**: Update operation terminates.
- **Strong convergence**: Correct replicas that have applied the same update have equivalent state.
How to implement CRDTs
State-based CRDTs: Counter

- Synchronization by propagating replica state
- Updates must inflate the state
- State must form a join semi-lattice wrt merge

⇒ Merge must be idempotent, commutative, associative
Join-semilattice

- A join-semilattice $S$ is a set that has a join (i.e. a least upper bound) for any non-empty finite subset:

For all elements $x, y \in S$, the least upper bound (LUB) $x \sqcup y$ exists.

- A semilattice is commutative, idempotent and associative.
- A partial order on the elements of $S$ is induced by setting $x \leq y$ iff $x \sqcup y = y$. 
Examples

\[
\begin{array}{c}
\{x,y,z\} \\
\{x,y\} & \{x,z\} & \{y,z\} \\
\{x\} & \{y\} & \{z\} \\
\emptyset
\end{array}
\]
Example: Counter

*Specification 6* State-based increment-only counter (vector version)

1: payload integer\([n]\) \(P\) 
2: \hspace{1cm} initial \([0, 0, \ldots, 0]\)
3: \hspace{1cm} update *increment* ()
4: \hspace{1cm} let \(g = myID()\)
5: \hspace{1cm} \(P[g] := P[g] + 1\) \hspace{1cm} ▶ \(g\): source replica
6: \hspace{1cm} query *value* () : integer \(v\)
7: \hspace{1cm} let \(v = \sum_i P[i]\)
8: \hspace{1cm} compare \((X, Y)\) : boolean \(b\)
9: \hspace{1cm} let \(b = (\forall i \in [0, n - 1] : X.P[i] \leq Y.P[i])\)
10: \hspace{1cm} merge \((X, Y)\) : payload \(Z\)
11: \hspace{1cm} let \(\forall i \in [0, n - 1] : Z.P[i] = \max(X.P[i], Y.P[i])\)
Operation-based CRDTs

- Concurrent updates must commute
- Requires reliable causal delivery for CRDTs with non-commutative operations
Example: Counter

**Specification 5** op-based Counter

1. payload integer \( i \)
2. initial 0
3. query *value* () : integer \( j \)
4. let \( j = i \)
5. update *increment* ()
6. downstream () \(\triangleright\) No precond: delivery order is empty
7. \( i := i + 1 \)
8. update *decrement* ()
9. downstream () \(\triangleright\) No precond: delivery order is empty
10. \( i := i - 1 \)
Example: Add-wins Set (Observed-remove Set)

\[
\begin{align*}
\{a\}.\text{add}(a) &= \{a, a\} & \text{unique} \\
\{a, b\}.\text{rmv}(a) &= \{a, b\} & \text{mark visible instances of } a \\
\{a, a\}.\text{contains}(a) &= \text{true} & \exists \text{ non-marked instance of } a \\
\{a, b\}.\text{merge}(\{a, c\}) &= \{a, b, c\} & \text{union + marker}
\end{align*}
\]
Example: Add-wins Set (Observed-remove Set)

payload set $E$, set $T$

initial $\emptyset, \emptyset$

query `contains` (element $e$) : boolean $b$
  let $b = (\exists n : (e, n) \in E)$

query `elements` () : set $S$
  let $S = \{ e | \exists n : (e, n) \in E \}$

update `add` (element $e$)
  prepare ($e$)
  let $n = \text{unique}()$
  effect ($e, n$)
  \[
  E := E \cup \{(e, n)\} \setminus T
  \]

update `remove` (element $e$)
  prepare ($e$)
  let $R = \{(e, n)|\exists n : (e, n) \in E\}$
  effect ($R$)
  \[
  E := E \setminus R \\
  T := T \cup R
  \]
Possible to garbage-collect the tombstone after remove

Trick: Assuming causal delivery, a removed element will never be re-introduced (with the same id)[2]
Optimized version of Add-wins Set

payload set $E$, vect $v$

\[ \text{initial } \emptyset, [0, \ldots, 0] \]

query contains (element $e$) : boolean $b$

\[ \text{let } b = (\exists c, i : (e, c, i) \in E) \]

query elements () : set $S$

\[ \text{let } S = \{e|\exists c, i : (e, c, i) \in E\} \]

update add (element $e$)

\[ \text{prepare } (e) \]

\[ \text{let } r = myID() \]

\[ \text{let } c = v[r] + 1 \]

\[ \text{effect } (e, c, r) \]

\[ \text{pre causal delivery} \]

\[ \text{if } c > v[r] \text{ then} \]

\[ \text{let } O = \{(e, c', r) \in E|c' < c\} \]

\[ v[r] := c \]

\[ E := E \cup \{(e, c, r)\} \setminus O \]

update remove (element $e$)

\[ \text{prepare } (e) \]

\[ \text{let } R = \{(e, c, i) \in E\} \]

\[ \text{effect } (R) \]

\[ \text{pre causal delivery} \]

\[ E := E \setminus R \]
Challenges with CRDTs

- Meta-data overhead for CRDTs that require causal contexts
  - Version vectors track concurrent modifications
  - Problematic under churn (i.e. when nodes come and go)
- Monotonically growing state with state-based approach
  - Infeasible for inherently growing data types such as sets, maps, lists with prevalent add
  - When removing elements, often tombstones are required for conflict resolution that relies on concurrency information
  - Requires garbage collection of tombstones when updates become causally stable
- Composability
  - CRDTs can be recursively nested (e.g. Maps, Sequences) or atomically updated in transactions
  - Which type of composability is preferable? What is the semantics of the composed entity?
Delta-based CRDTs

- State-based CRDTs suffer from monotonically growing state (lattice!)
- Op-based CRDTs require reliable causal delivery

Delta-based CRDTs[1]

- Small message comprising a set of incremental updates
- Works over unreliable communication channels (because idem-potent and commutative)

A delta-mutator $m^\delta$ is a function, corresponding to an update operation, which takes a state $X$ in a join-semilattice $S$ as parameter and returns a delta-mutation $m^\delta(X)$, also in $S$.

$$X' = X \sqcup m^\delta(X)$$
Adoption of CRDTs in industry
Conclusion

- CRDTs provide Strong Eventual Consistency (sometimes even more)
- Properties of good conflict resolution
  - Don’t lose updates/information!
  - Deterministic (independent of local update order)
  - Semantics close to sequential version
- Meta-data overhead can be substantial
Further reading I

