Operation-based CRDTs

Paulo Sérgio Almeida

Universidade do Minho

Conflict-free Replicated Data Types

Operation-based CRDTs

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 operation-based CRDTs

Conflict-free Replicated Data Types

Operation-based CRDTs

Operation-based CRDTs

- Core concept
  - send operations, not state, to other replicas
  - operations applied at each replica
- Uses reliable causal broadcast
  - ensures exactly once for non-idempotent operations
  - respects order of causally dependent operations
- What about non-commutative operations?
  - applied in different orders would lead to divergence
  - dealt with by sending more than just the operation

Standard execution model of op-based CRDTs

- Prepare performed at replica where operation is invoked
  - looks at state and op
  - does not have side effects (on abstract state)
  - returns message to be sent
- Message disseminated with reliable causal broadcast
- Upon message delivery, effect is applied at each replica
  - assumes immediate self-delivery on sender replica
- Effect designed to be commutative for concurrent ops
  - ensures convergence under different application orders

Simple CRDTs – with commutative and associative ops

- If the operations are associative and commutative ...
  ...they can be grouped and applied in any order
- Examples:
  - Counter
  - Positive-Negative Counter (PN-Counter)
  - Grow-only set (GSet)
- Such op-based CRDTs are trivial, given exactly-once delivery
- Not even FIFO is needed for convergence
- Causal delivery normally used to achieve causal consistency

Counter

CRDT state:
$n : \mathbb{N} = 0$

query value() : $\mathbb{N}$
return n

update inc()
prepare
return inc
effect inc
$n \leftarrow n + 1$

Positive-negative counter (PNCounter)

CRDT state:
$v : \mathbb{Z} = 0$

query value() : $\mathbb{Z}$
return v

update inc()
prepare
return inc
effect inc
$v \leftarrow v + 1$

update dec()
prepare
return dec
effect dec
$v \leftarrow v - 1$

Grow-only set (Gset$\langle E \rangle$)

CRDT state:
$s : \mathcal{P}(E) = \emptyset$

query elements() : E
return s

query contains($e : E$) : $\mathbb{B}$
return $e \in s$

update add($e : E$)
prepare
return (add, e)
effect (add, e)
$s \leftarrow s \cup \{e\}$

CRDTs with non-commutative operations

- Example: set with add and remove operations:

$$\mathsf{add}(v, \mathsf{rmv}(v, s)) \neq \mathsf{rmv}(v, \mathsf{add}(v, s))$$

- Concurrent operations will be delivered in different orders
- Applying them to a sequential datatype would cause divergence
- CRDT cannot be simply the sequential datatype
- Effect must be defined to be commutative for concurrent ops

Defining concurrent semantics

- We must define how to handle conflicts
- Example: given a concurrent add and remove, which will "win"
- In general many options possible
- An elegant concept: act on observed (visible) ops

> If some op cancels others, do it only to observed ops

- Example: an observed-remove set
  - has add and remove ops
  - a remove only cancels adds that are visible to it
  - with causal delivery, cancels adds in the causal past
  - concurrent adds will not be canceled, and will "win"
- Used elsewhere:
  - an observed-reset counter: resets cancel observed incs

Observed-remove set (ORSet$\langle E \rangle$), vanilla, sketch

CRDT state:
$s : \mathcal{P}(E \times \dots) = \emptyset$

query elements() : E
return $\{e \mid (e, \_) \in s\}$

query contains($e : E$) : $\mathbb{B}$
return $\exists x \cdot (e, x) \in s$

update add($e : E$)
prepare
let $u =$ [some unique id]
return (add, e, u)
effect (add, e, u)
$s \leftarrow s \cup \{(e, u)\}$

update remove($e : E$)
prepare
let $r = \{(x, u) \in s \mid x = e\}$
return (rmv, r)
effect (rmv, r)
$s \leftarrow s \setminus r$

- Assumes reliable causal delivery

Observed-remove set, issues, improvements

- Sketch assumes generation of unique ids; how?
  - can use counter per replica, incremented at each add
  - unique id obtained as pair (replica id, counter)
  - sometimes called a “dot” (from Dotted Version Vectors)
  - counter not used by queries or effect; auxiliary state
- Most operations need set traversal
  - solution: use a map from elements to sets of ids
- Adds accumulate entries, even if element already present
  - solution: replace current entries for given element
- In remove, prepare sends element repeated in each pair
  - solution: collect set of ids separately

Observed-remove set (ORSet$\langle E \rangle$), optimized

types:
$\mathbb{I}$, set of replica identifiers

CRDT state:
$m : E \rightharpoonup \mathcal{P}(\mathbb{I} \times \mathbb{N}) = \emptyset$
$c : \mathbb{N} = 0$, auxiliary state

query elements() : E
return dom m

query contains($e : E$) : $\mathbb{B}$
return $m[e] \neq \emptyset$

update add($e : E$)
prepare
$c \leftarrow c + 1$
return $(\text{add}, e, (i, c), m[e])$
effect (add, e, d, r)
$m[e] \leftarrow m[e] \setminus r \cup \{d\}$

update remove($e : E$)
prepare
return $(\text{remove}, e, m[e])$
effect (remove, e, r)
$m[e] \leftarrow m[e] \setminus r$

- Algorithm for replica $i \in \mathbb{I}$
- Assumes reliable causal delivery
- Map stores only non-"bottom" values (non-empty sets)
- Map returns $\emptyset$ for unmapped keys

CRDTs non-equivalent to serialized execution

- Most CRDTs resolve conflicts aiming to achieve behavior equivalent to some sequential executiom
- In some cases the CRDT:
  - has behavior not possible by any sequential execution
  - the interface itself is different from sequential datatype
- Most well know case: multi-value register
  - made popular by the Amazon Dynamo paper
  - register keeps set of most recent concurrent writes
  - a read returns that set
  - a write overwrites the set into a singleton

Example: multi-value register (MVReg$\langle E\rangle$)

types:
$\mathbb{I}$, set of replica identifiers

CRDT state:
$s : \mathcal{P}(E \times (\mathbb{I} \times \mathbb{N})) = \emptyset$
$c : \mathbb{N} = 0$, auxiliary state

query read() : $\mathcal{P}(E)$
return $\{e \mid (e, \_) \in s\}$

update write($e : E$)
prepare
$c \leftarrow c + 1$
let $r = \{d \mid (\_, d) \in s\}$
return $(\mathrm{write}, (e, (i, c)), r)$
effect (write, v, r)
$s \leftarrow \{(e, d) \in s \mid d \notin r\} \cup \{v\}$

- Algorithm for replica $i \in \mathbb{I}$
- Assumes reliable causal delivery