Conflict-free Replicated Data Types

Wojciech Grabis

May 21, 2020

Introduction

- Replication and consistency are essential features of any large distributed system.
- Synchronisation requires to implement some form of conflict resolution.
- CRDT allows to achieve synchronisation without any conflict resolution, using a simple theoretical model.

Strong eventual consistency

Definition (Eventual Consistency)

Eventual delivery An update delivered at some correct replica is eventually delivered to all correct replicas.
Convergence Correct replicas that have delivered the same updates eventually reach equivalent state.
Termination All method executions terminate.

Definition (Strong eventual consistenct)

An object is Strongly Eventually Consisten if it is Eventually Consistent and
Strong Convergence Correct replicas that have delivered the same updates have equivalent state.

System model

Strong Eventual Consistency system model presented proposes two styles of replicated objects:

- State-based object
- Operation-based object

We define the object as a tuple (S, s^0, q, u, m).

- Each replica process p_i has state $s_i \in S$ called payload.
- Initial state is s^0
- Each client can read using query method q
- Updates happen through method u
- Merging is done using method m

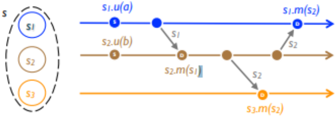

Fig. 1. State-based replication

State-based object

Definition (State-based Causal History)

We define the object's causal history $C = [c_1, \dots, c_n]$, where c_i is a sequence of states for process p_i $c_i^0, \dots, c_i^k, \dots$. Initially $c_i^0 = \varnothing$. If the k^{th} method execution at i is

- a query q: history does not change $c_i^k = c_i^{k-1}$
- an update $u_i^k(a)$: is is added to history $c_i^k = c_i^{k-1} \bigcup \{u_i^k(a)\}$
- a merge $m_i^k(s_{i'}^{k'})$: then the local and remote histories are unioned together $c_i^k = c_i^{k-1} \bigcup c_{i'}^{k'}$

State-based Convergent Replicated Data Type (CvRDT)

Definition (Monotonic semilattice object)

A state-based object equipped with partial order $\leq$, noted $(S, \leq, s^0, q, u, m)$, that has the following properties, is called monotonic semi-lattice:

- Set S of payload values forms a semilattice ordered by $\leq$
- Merging state s with remote state s' computes the least upper bound of the two states, i.e., $s \bullet m(s') = s \sqcup s'$
- State is monotonically non-decreasing across updates, i.e., $s \leq s \bullet m(s')$

Theorem (Convergent Replicated Data Type)

Assuming eventual delivery and termination, any state-based object that satisfies the monotonic semilattice property is SEC.

Operation-based object

Operation-based object is a tuple (S, s^0, q, t, u, P).

- Definition of S, s^0, q stays the same
- Update method is split into a pair (t, u), where t is a side-effect-free *prepade-update* method and u is an *effect-update* method.
- Prepare-update method is followed immediately by effect-update method, effect-update method executes on all replicas.
- Additionaly we specify a delivery relation P, called delivery precondition, an effect-update method is enabled only if $P(s_i, u)$ is true

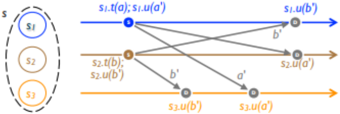

Fig. 2. Operation-based replication

Operation-based object

Definition (Operation-based Causal History)

An object's causal history $C = \{c_1, \ldots, c_n\}$ is defined as follows. Initially, $c_i^0 = \emptyset$, for all i. If the k^{th} method execution is

- a query q or a prepade update t, the casual history does not change $c_i^k = c_i^{k-1}$
- an effect-update $u_i^k(a)$, then $c_i^k = c_i^{k-1} \bigcup \{u_i^k(a)\}$

Definition (Concurrent updates)

An update is said delivered at a replica, when the update is included in the replica's causal history. Update (t, u) happened before (t', u'):
$(t, u) \rightarrow (t', u') \iff u \in c_j^{k-1}$, where t' executes at p_j in step k.
Updates are *concurrent* $u \| u' = u \not\rightarrow u' \wedge u' \not\rightarrow u$

Commutative Replicated Data Type (CmRDT)

Definition (Commutativity)

Updates (t, u) and (t', u') commute, iff for any reachable replica state s where both u and u' are enabled, u remains enabled in state $s \bullet u'$ and $s \bullet u \bullet u' \equiv s \bullet u' \bullet u$ (respectively u')

Theorem (Commutative Replicated Data Type (CmRDT))

Assuming casual delivery of updates and method termination, any op-based object that satisfies the commutativity property for all concurrent updates, and whose delivery precondition is satisfied by casual delivery, is SEC.

State and operation based equivalency

Theorem

Any SEC state-based object can be emulated by a SEC operation-based object of a corresponding interface.

Theorem

Any SEC operation-based object can be emulated by a SEC state-based object of a corresponding interface.

Example - integer vector

We consider state-oriented object of a vector-of-integers $(\mathbb{N}^n, [0,\ldots,0], \leq^n, [0,\ldots,0], value, inc, max^n)$.

- Vectors v, $v' \in \mathbb{N}^n$ are partially ordered by $v \leq^n v' \iff \in [0,\ldots,n-1] : v[i] \leq v'[i]$
- As update we treat function $inc(i)$, which increments the payload value at index i
- Merging two vectors take the maximum for each index, so $s \bullet max^n(s') = [max(s[0], s'[0]), \ldots, max(s[n-1], x'[n-1])]$.

Example - U-Set

For set an implementation using two add-only sets (A, R) can be proposed.

- Operation $add(e)$ adds an element to set A, while operation $remove(e)$ adds element to set R, if it exists in set A.
- For $value()$ we return $A \setminus R$

Direct graph structure

We consider a graph representing a structure of web pages for web crawlers.

- Each process using will process edges of the graph, representing a web page link.
- When a crawler finds a new page, it executes *addVertex* and compares the version with previous, execurint *addArc* and *removeArc* for corresponding links.

payload set V, A -- *sets of pairs* $\{$ (element e, unique-tag w), $\ldots\}$

    initial $\varnothing, \varnothing$ -- V*: vertices;* A*: arcs*

query $lookup$ (vertex v) : boolean b

    let $b = (\exists w : (v, w) \in V)$

query $lookup$ (arc (v', v'')) : boolean b

    let $b = (lookup(v') \wedge lookup(v'') \wedge (\exists w : ((v', v''), w) \in A)$

update $addVertex$ (vertex v)

    prepare (v) : w

        let $w = unique()$ -- $unique()$ *returns a unique value*

    effect (v, w)

        $V := V \cup \{(v, w)\}$ -- v + *unique tag* update $removeVertex$ (vertex v)

prepare (v) : R

pre $lookup(v)$ -- *precondition*

pre $\nexists v' : lookup((v, v'))$ -- *v is not the head of an existing arc*

let $R = \{(v, w) | \exists w : (v, w) \in V\}$ -- *Collect all unique pairs in V containing v*

effect (R)

$V := V \setminus R$

update $addArc$ (vertex v', vertex v'')

prepare (v', v'') : w

pre $lookup(v')$ -- *head node must exist*

let $w = unique()$ -- *$unique()$ returns a unique value*

effect (v', v'', w)

$A := A \cup \{((v', v''), w)\}$ -- *(v', v'') + unique tag*

update $removeArc$ (vertex v', vertex v'')

prepare (v', v'') : R

pre $lookup((v', v''))$ -- *$arc(v', v'')$ exists*

let $R = \{((v', v''), w) | \exists w : ((v', v''), w) \in A\}$

effect (R) -- *Collect all unique pairs in A containing arc (v', v'')*

$A := A \setminus R$

Fault tolerance

- SEC replica is always available for both reads and writes
- Any subset of replicas of a SEC object eventually converges
- There is no consensus required, so SEC object tolerates up to $n - 1$ crashes of replica processes.