Strong Eventual Consistency and CRDTs

Marc Shapiro, INRIA & LIP6
Nuno Preguiça, U. Nova de Lisboa
Carlos Baquero, U. Minho
Marek Zawirski, INRIA & UPMC
Large-scale replicated data structures

Wish list:
• Mutable
• Incremental
• Fast $\Rightarrow$ parallel, asynch
• Fault tolerant

Eventual Consistency
• Principles?
**Strong consistency**

- Preclude conflict: Replicas update in same total order
- Any deterministic object
- Consensus
  - Serialisation bottleneck
  - Tolerates $< n/2$ faults
- Sequential, linearisable...
- *Universal*
Eventual Consistency

Update local + propagate
• No foreground synch
• Expose tentative state
• Eventual, reliable delivery

On conflict
• Arbitrate
• Roll back

Consensus moved to background
Strong Eventual Consistency

Update local + propagate
• No synch
• Expose intermediate state
• Eventual, reliable delivery

No conflict
• Unique outcome of concurrent updates

No consensus: \( \leq n-1 \) faults
Not universal
Fast, responsive
Solves the CAP problem
**Strong Eventual Consistency**

**Eventual delivery:** An update executed at some correct replica eventually executes at all correct replicas.

**Termination:** All update executions terminate.

**Strong Convergence:** Correct replicas that have executed the same updates have equivalent state.

- No conflicts
- No rollback
- No consensus
- Limited
Conflict-free Replicated Data Types (CRDTs)

Intuition:
- *Conflict resolution* requires synchronisation
- Conflict-freedom satisfies SEC

⇒ Design data types with no conflicts

CRDTs
- Available, fast
- Reconcile scalability + consistency

Simple sufficient conditions
- Principled, correct
State-based replication

Local at source $s_1.u(a), s_2.u(b), \ldots$
- Compute
- Update local payload

Convergence:
- Episodically: send $s_i$ payload
- On delivery: merge payloads $m$
State-based: monotonic semi-lattice ⇒ CRDT

If

- payload type forms a semi-lattice
- updates are increasing
- *merge* computes Least Upper Bound

then replicas converge to LUB of last values

Example: Payload = int, *merge* = *max*
Operation-based replication

At source:
- prepare
- broadcast to all replicas

Eventually, at all replicas:
- update local replica

• push to all replicas eventually
• push small updates
  - more efficient than state-based
Operation-based replication

At source:
• prepare
• broadcast to all replicas

Eventually, at all replicas:
• update local replica

Strong Eventual Consistency
Op-based: commute $\Rightarrow$ CRDT

If: • (Liveness) all replicas execute all operations in delivery order
   • (Safety) concurrent operations all commute
Then: replicas converge
Composition and sharding

A composition of independent CRDTs is a CRDT

Very large objects
- Independent *shards*
- Static: hash

Statically-Sharded CRDT
- Each shard is a CRDT
- Update: single shard
- No cross-object invariants

(Dynamic: requires consensus to rebalance)
The challenge:
What interesting objects can we design with no synchronisation whatsoever?
Portfolio of CRDTs

Register
- Last-Writer Wins
- Multi-Value

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

Map
- Set of Registers

Counter
- Unlimited
- Non-negative

Graphs
- Directed
- Monotonic DAG
- Edit graph

Sequence
- Edit sequence
Multi-master counter

Increment / decrement

- Payload: $P = [\text{int, int, } \ldots], \quad N = [\text{int, int, } \ldots]$

- $\text{value()} = \sum_i P[i] - \sum_i N[i]$

- $\text{increment()} = P[\text{MyID}]++$

- $\text{decrement()} = N[\text{MyID}]++$

- $\text{merge}(s,s') = (\ldots,\max(s.P[i],s'.P[i]),\ldots)_{i}, (\ldots,\max(s.N[i],s'.N[i]),\ldots)_{i}$

- Positive or negative
Multi-master counter

Increment / decrement

- Payload: \( P = [\text{int, int, ...}] \), \( N = [\text{int, int, ...}] \)
- \( \text{value}() = \sum_i P[i] - \sum_i N[i] \)
- \( \text{increment}() = P[\text{MyID}]++ \)
- \( \text{decrement}() = N[\text{MyID}]++ \)
- \( \text{merge}(s,s') = s \sqcup s' = ([...,
  \max(s.P[i],s'.P[i]),...],
  [...,
  \max(s.N[i],s'.N[i]),...])_i \)
- Positive or negative

"like vector clock"

"can't maintain global invariant such as s>0"
Set design alternatives

Sequential specification:
- \{true\} add(e) \{e ∈ S\}
- \{true\} remove(e) \{e ∉ S\}

\{true\} add(e) || remove(e) \{????\}
- linearisable?
- error state?
- last writer wins?
- add wins?
- remove wins?

• linearisable: sequential order
• equivalent to real-time order
• Requires consensus
2P-Set

Payload = (Grow-Set \( A \), Grow-Set \( R \))
- \( add (e) = A := A \cup \{e\} \)
- \( remove (e) = e \in A ? R := R \cup \{e\} \)
- \( lookup (e) = e \in A \land e \notin R \)
- \( s \leq s' \defeq s.A \subseteq s'.A \land s.R \subseteq s'.R \)
- \( merge (s,s') = (s.A \cup s'.A, s.R \cup s'.R) \)

\{true\} add(e) || remove(e) \{e \notin S\}
**Observed-Remove Set**

- **Payload**: added, removed (element, unique-token)
  \[ \text{add}(e) = A \triangleq A \cup \{(e, \alpha)\} \]
- **Remove**: all unique elements observed
  \[ \text{remove}(e) = R \triangleq R \cup \{(e, -) \in A\} \]
- **lookup**: \( \exists (e, -) \in A \setminus R \)
- **merge**: \( (S, S') = (A \cup A', R \cup R') \)
- **{true} add(e) || remove(e) \{e \in S\}**

- Can never remove more tokens than exist
- Op order \( \rightarrow \) removed tokens have been previously added
OR-Set

Set: solves Dynamo Shopping Cart anomaly

Optimisations:
• No tombstones
• Operation-based approach
• Snapshots
• Sharded
OR-Set + Snapshot

Read consistent snapshot
• Despite concurrent, incremental updates

Unique token = time (vector clock)
• $\alpha = $ Lamport ($process\ i,\ counter\ t$)
• UIDs identify snapshot version
• Snapshot: vector clock value
• Retain tombstones until not needed

$$\text{lookup}(e, t) = \exists (e, i, t') \in A : t' > t \land \nexists (e, i, t') \in R : t' > t$$
OR-Set + Snapshot (2)

- Payload: vector clock $V_i$
  
  set $A_i = \{ (e, j, t), \ldots \}$
  
  set $R_i = \{ (e, j, c, j', t'), \ldots \}$

- $\text{add}(e): V_i[i]++; A_i := A_i \cup \{ (e, i, V_i[i]) \}$

- $\text{remove}(e): V_i[i]++; R_i := R_i \cup \{ (e, j, t, i, V_i[i]) \}$

- $\text{merge}(V, A, R)$:
  
  $\forall j, V_i[j] := \max(V_i[j], V[j])$; $A_i := A_i \cup A$; $R_i := R_i \cup R$

- $\text{lookup}(e, V)$:
  
  $\exists (e, j, c) \in A_i \land (e, j, c, –, –) \notin R_i \land V[j] \geq c$

  $\lor (e, j, c, j', c') \in R_i \land V[j] \geq c \land V[j'] < c$
Graph design alternatives

Graph = (V, E) where E ⊆ V × V

Sequential specification:
• \( \{v, v' \in V\} \) addEdge\((v, v')\) \{…\}
• \( \{\nexists (v, v') \in E\} \) removeVertex\((v)\) \{…\}

Concurrent: removeVertex\((v')\) \| addEdge\((v, v')\)
• linearisable?
• last writer wins?
• addEdge wins?
• removeVertex wins?
• etc.

*for our Web Search Engine application, removeVertex wins*
*Do not check precondition at add/remove*
Payload = OR-Set $V$, OR-Set $E$

Updates add/remove to $V$, $E$
- $\text{addVertex}(v)$, $\text{removeVertex}(v)$
- $\text{addEdge}(v,v')$, $\text{removeEdge}(v,v')$

Do not enforce invariant a priori
- $\text{lookupEdge}(v,v') = (v,v') \in E$
- $v \in V \land v' \in V$

$\text{removeVertex}(v') \parallel \text{addEdge}(v,v')$
- removeVertex wins
Graph + shards + snapshots

Snapshot
- see OR-Set

Sharding
- See OR-Set
- Do not enforce invariant \( a \ priori \)
  \[
  \text{lookupEdge}(v,v') = (v,v') \in E \\
  \land v \in V \land v' \in V
  \]
CRDT + dataflow

Incremental, asynchronous processing
• Replicate, shard CRDTs near the edge
• Propagate updates ≈ dataflow
• Throttle according to QoS metrics
  (freshness, availability, cost, etc.)

Scale: sharded
Synchronous processing: snapshot, at centre
Thought experiment (2)

- content DB, index: decentralised, replicated, close to the network edge
Contributions

Strong Eventual Consistency (SEC)
• A solution to the CAP problem
• Formal definitions
• Two sufficient conditions
• Strong equivalence between the two
• SEC shown incomparable to sequential consistency

CRDTs
• integer vectors, counters
• sets
• graphs