From strong to eventual consistency: getting it right

Nuno Preguiça, U. Nova de Lisboa
Marc Shapiro, Inria & UPMC-LIP6
Conflict-free Replicated Data Types *

Marc Shapiro\textsuperscript{1,5}, Nuno Preguiça\textsuperscript{2,1}, Carlos Baquero\textsuperscript{3}, and Marek Zawirski\textsuperscript{1,4}

\textsuperscript{1} INRIA, Paris, France
\textsuperscript{2} CITI, Universidade Nova de Lisboa, Portugal
\textsuperscript{3} Universidade do Minho, Portugal
\textsuperscript{4} UPMC, Paris, France
\textsuperscript{5} LIP6, Paris, France

Abstract. Replicating data underlying a Web service ensures performance and scalability in large-scale distributed systems. However, published EC approaches are ad-hoc and error-prone. This paper formalises two popular approaches (state- and operation-based) and their relevant guarantees. Building reliable distributed systems requires performance and scalability in large-scale distributed systems (e.g., clouds). However, published EC approaches are ad-hoc and error-prone. This paper formalises two popular approaches (state- and operation-based) and their relevant guarantees.

Keywords: Eventual Consistency, Replicated Shared Objects, Large-Scale Distributed Systems

A broader class of consistency guarantees can, and perhaps should, be offered to clients that read shared data.

BY DOUG TERRY

Replicated Data Consistency Explained Through Baseball

review articles
Strong consistency is easy to understand, but it requires synchronisation, which has a high cost and does not tolerate partial system failure or network partitions well. Hence, the past decade has seen renewed interest in an alternative - Eventual Consistency (EC) - where data is accessed and modified without coordination. Many production systems guarantee only EC, including NoSQL databases, key-value stores and cloud storage systems. EC improves responsiveness and availability, updates propagate more quickly, and enables less coordination-intensive and often less complex protocols.

EC is crucial for scalability but represents an unfamiliar design space. Conflicts and divergence are possible, making them hard to program against, while metadata growth and maintaining system-wide invariants can also become problematic. As more developers interact with and program against distributed storage systems, understanding and managing these trade-offs becomes increasingly important.

Although EC remains poorly understood, there has recently been considerable momentum in the research and development community: the past several years have seen the (re)introduction of useful concepts, such as replicated data types, monotonic programming, causal consistency, red-blue consistency, novel proof systems, etc., designed to allow improve efficiency, programmability, and overall operation of weakly consistent stores.

This workshop aims to investigate the principles and practice of Eventual Consistency. It will bring together theoreticians and practitioners from different horizons: system development, distributed algorithms, concurrency, fault tolerance, databases, language and verification, including both academic and industry. In order to make EC computing easier and more accessible, and to address the limitations of EC and explore the design space spectrum between EC and strong consistency, we will share experience with practical systems and from theoretical study, to investigate algorithms, design patterns, correctness conditions, complexity results, and programming methodologies and tools.

Relevant discussion topics include:

- Design principles, correctness conditions, and programming patterns for EC.
- Techniques for eventual consistency: session guarantees, causal consistency, operational transformation, conflict-free replicated data types, monotonic programming, state merge, commutativity, etc.
- Consistency vs. performance and scalability trade-offs: guiding developers, controlling the system.
- Analysis and verification of EC programs.
- Strengthening guarantees: transactions, fault tolerance, security, ensuring invariants, bounding metadata memory, and controlling divergence, without losing the benefits of EC.
- Platform guarantees vs. application involvement: guiding developers, controlling the system.

Workshop format

The workshop will feature a small number of invited talks. Each workshop session will contain a number of presentations around a common theme, followed by a panel and a general discussion on the theme.

Paper submission

We solicit proposals for contributed talks. We recommend preparing proposals of at most 2 pages, in either
From Strong to Eventual Consistency

Strong vs. eventual consistency
Pros and cons
Historical perspective
Examples: Facebook, file systems, Bayou
data centre
Strongly-consistent replication

synchronisation

synchronisation

synchronisation
Asynchronous propagation

California

Bob

Martin

Nellie

Paris

Bob

Nellie

Alice

Martin

Availability
Responsiveness
Performance
No congestion

Conflict!
Eventual consistency

From strong to eventual consistency: getting it right
Why bother?

Latency [ms]

Throughput [TPM x 1000]

classic synch.

SwiftCloud
EC Historical perspective (1)

Different backgrounds, requirements, approaches

Slow, error-prone networks
- uunet $\rightarrow$ epidemic communication
- DNS: Thomas Write Rule: LWW
- Wuu & Bernstein [PODC 1984]

Mobile computing $\rightarrow$ opportunistic communication, algorithmic conflict resolution
- File systems: Coda, Locus/Ficus, resolvers
- Bayou: tentative execution & rollback
- IceCube: app-independent
Group editing, source code management
- Offline, textual merge: CVS, SVN, git
- Online, operation-based: Operational Transformation (OT), Google Drive

P2P and cloud computing: fast, deterministic, program-oriented
- NoSQL, DHTs, key-value stores: LWW, Dynamo, C-set
- Causal consistency: Walter, Cops/Eiger
Example: Facebook

- Read from closest server
- Write to California
- Other servers update cache every 15 minutes
- After write: read from CA for 15 minutes
Example: Replicated files

Update: assign, overwrite file
- Transmit file + timestamp (unique, monotonic)
- Highest timestamp wins

Widely used: file systems, KVSs

[Johnson, Thomas, *The maintenance of duplicate databases*, 1976, RFC 677]
Example: Bayou

Arbitrary database, code
Update = ( dep-check, update-proc, merge-proc )
  • Execute, append to log

Anti-entropy
  • Send My-Log \ Remote-Log to peer

Receive update
  • Order by timestamp
  • Possibly: Roll back, re-execute

Convergence: primary site imposes order
  • Consensus by dictatorship

Prune logs
Basic Mechanisms

Read and update replica (multi-master)
Transmit later
Detect order / concurrency / conflict
Reconcile: converge to a common state
Garbage-collect metadata
multiple replicas: ¬ monotonic reads

Query

Query local replica

Bayou: read database copy
Update source replica
Transmit to *downstream* replicas later
- In background
- Flooding, anti-entropy, etc.
Receiver applies update

multiple replicas: ¬ Read My Writes
Merge two (or three) states:
  • How to ensure convergence?

Delivery:
  • P2P "epidemic," probabilistic
  • Star

File systems: dropbox, SVN

Inefficient for large payload
Convergence?

Epidemic ⇒ probabilistic event delivery
Op-based / function shipping

- Reliable broadcast
- Different execution orders
  - How to ensure convergence?

Bayou, IceCube, OT
Op-based / function shipping

• Reliable broadcast
• Different execution orders
  ▸ How to ensure convergence?

Bayou, IceCube, OT
Op-based / function shipping

- Reliable broadcast
- Different execution orders
  - How to ensure convergence?
  
  *Bayou, IceCube, OT*
Vector clocks & anti-entropy

\[ V_i[j] = \# \text{events of } j \text{ observed by } i \]

\[ V_i[j] - V_i'[j] = \text{events of } j \text{ that } i' \text{ should transmit to } i \]

\[ V(a) < V(b) = a \text{ happened before } b \]

\[ V(a) \prec V(d) \land V(d) \prec V(a) = a \text{ concurrent with } b \]
Detect concurrent updates

Two updates:
• If \( u < v \), then \( v \)
• If \( (u \preceq v \land v \preceq u) \), then \( \) ????

Detect concurrency
• Explicit history/dependence (\textit{Isis}, \textit{Cops/Eiger})
• Vector clocks (\textit{Isis 2}, \textit{SwiftCloud})
• Programmed check (\textit{LWW}, \textit{Bayou}, \textit{Sinfonia})

Requires meta-data
Vector clocks & concurrency

\[ V_i[j] = \# \text{events of } j \text{ observed by } i \]
\[ V_i[j] - V_i'[j] = \text{events of } j \text{ that } i' \text{ should transmit to } i \]
\[ V(a) < V(b) = a \text{ happened before } b \]
\[ V(a) \prec V(d) \land V(d) \prec V(a) = a \text{ concurrent with } b \]
Resolve concurrent updates

Resolve:
- No conflict
- Dynamic total order (*Bayou, IceCube*)
  ‣ Consensus!
- Static total order: arbitration (*LWW*)
- Resolver program (*SVN, Bayou*)
- Ask user

Convergence?
- All replicas make equivalent
Garbage-collect metadata

At least-once delivery:
- An update $u$ must be available for forwarding until received by all replicas

Wuu & Bernstein 1984
- Receiver $i$ sends $ack(u,i)$ to all other replicas
- $j$ received $ack(u,i)$ from $\forall i :$ discards $u$
- Not live if partitioned

Bayou
- Truncate log arbitrarily
- If necessary, recover with full-state transfer
- May lose updates
Distributed resolution

Bayou/IceCube conflict resolution:
• Convergence $\implies$ unique $\implies$ consensus
• Off critical path: availability OK
• Roll back: expensive, confusing

Alternative: decentralised resolution
• No synchronisation
• Convergence conditions:
  ‣ Deterministic
  ‣ Dependent on delivery set
  ‣ Not on delivery order, local info
Decentralised resolution:

**Highest timest. wins (LWW)**

Update: assign, overwrite file

- Transmit file + timestamp (*unique, monotonic*)
- Highest timestamp wins

Widely used: file systems, KVSs

[Johnson, Thomas, *The maintenance of duplicate databases*, 1976, RFC 677]
Decentralised resolution:

Multi-Value Register

Not concurrent: usual register semantics
Concurrent assignments: return set of values
Decentralised resolution:  

**2P-Set**

 Representation:  
- $A$: set of added items  
- $T$: set of removed items (tombstones)

 Operations:  
- $\text{add} (e)$  
  $$A := A \cup \{ e \}$$  
- $\text{rmv} (e)$  
  $$T := T \cup \{ e \}$$  
- $\text{lookup} (e)$  
  $$e \in A \setminus T$$

 Resolving concurrent updates:  
- $A := A_1 \cup A_2$  
- $T := T_1 \cup T_2$
Decentralised resolution:

Operational Transformation

Ensure convergence in any order
Transform received operations according to previous concurrent updates
OT correctness conditions

TP1: \[ f \cdot \frac{g}{f} \equiv g \cdot \frac{f}{g} \]

TP2: \[ \frac{h}{f \cdot \frac{g}{f}} = \frac{h}{f \cdot \frac{g}{f}} \]

Star topology: TP1
P2P topology: TP1 + TP2
TP1 \equiv commutativity
Basic Mechanisms

Read and update replica (multi-master)
  Transmit later
  Detect order / concurrency / conflict
Reconcile: converge to a common state
Garbage-collect metadata
Formal definitions

System model
Define EC
Partial Order model
Stronger Guarantees
Assume some shared objects + specification

Objects support query and update operations. Operations terminate.

Objects are replicated at a number of processes.

A client may update a replica:
- Update invocation returns
- Update is sent to other replicas
- Receiving replica applies the update
- Subject to correctness conditions hereafter
Eventual Consistency?

“If no new updates are made to the object, eventually all accesses will return the last updated value”

Captures intuition, but
• Informal
• Assumes update = overwrite
• What happens if updates don't stop?
Eventual Consistency

• Every update eventually reaches every replica at least once
• An update has effect on a replica at most once
• At all times, the state of a replica satisfies the objects’ specifications
• Two replicas that received the same set of updates eventually reach the same state

Transactional: replace “update” with “transaction comprising one or more updates”.
Asynch ∩ stronger guarantees

Session guarantees
- Read Your Writes
- Monotonic Reads, Monotonic Writes
- Writes Follow Reads

Causal consistency: “If v is visible, and u precedes v, then u is visible”
- Implies session guarantees
- Available ⟹ Strongest possible [MAD 2011]

Mergeable transactions
Fault tolerance

How to implement at scale?
Causal consistency

$x$ observed effects of $u$ at source
$\implies x$ to be delivered after $u$

Challenge: implement at scale
- Meta-data explosion
Causal consistency

x observed effects of u at source
⇒ x to be delivered after u

Challenge: implement at scale
• Meta-data explosion
Partial-Order model

[ Burckhardt, Gotsman, Yang, "Understanding Eventual Consistency", MSR-TR-2013-39 ]

Effect of $op = F(op, events, po, vis, ar)$

- $vis^{-l}$: Recent updates
- $ar$: Arbitrate concurrent updates
- $F$: Deterministic function
Result of an action?
Only visible actions matter
P-O model: LWW

Result of an action?
Only visible actions matter
P-O model: Guarantees

\( x := 3 \)  
\( (1, 1) \)

\( x := 4 \)  
\( (1, 3) \)

\( \text{hb} \overset{\text{def}}{=} (\text{po} \cup \text{vis})^+ \)

Read Your Writes: \( \text{po} \subseteq \text{vis} \)

Monotonic Reads: \( (\text{vis}; \text{po}) \subseteq \text{vis} \)

Causal: \( \text{hb} \subseteq \text{vis} \land \text{hb} \subseteq \text{ar} \)

Atomic + Isolated: ???

\( \Rightarrow \) RYW, MR, ...

[From strong to eventual consistency: getting it right]
Formal definitions

System model
Define EC
Partial Order model
Stronger Guarantees
CRDTs: Conflict-Free Replicated Data Types

Definition

Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs
Conflict-Free Replicated Data
Type: key concepts

Replicated at multiple nodes
Conflict-free: update without coordination, decentralised resolution

*Data type: Encapsulation, well-defined interface*

*Eventual convergence by construction*
Library of CRDTs

Register
- Last-Writer Wins
- Multi-Value

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

Map

File-system tree

Counter
- Unlimited
- Non-negative

Graph
- Directed
- Monotonic DAG
- Edit graph

Sequence
State-based CRDT

Convergence?
Safety requirements:
- Don’t go backwards ⇒ partial order, monotonic
- Apply each update once: \( m \) idempotent
- Merge in any order: \( m \) commutative
- Merge contains several updates: \( m \) associative

*Monotonic semi-lattice + merge = Least Upper Bound*
Example: highest timestamp wins register

State:

\[ V_{ts} : \text{value } V, \text{ timestamp } ts \]

Operations:

- **write**: overwrite \( v \), increment \( ts \)
- **merge**: value with highest \( ts \) wins

\[ s \leq s' \iff s.ts \leq s'.ts \]

\[ \text{merge} \ (s, s') = s.t < s'.t \ ? s' : s \]
CRDTs: Conflict-Free Replicated Data Types

Definition

Convergence conditions: State-vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs
Operation-based CRDT

Operation/function shipping
Reliable broadcast

Convergence?
Convergence: sufficient condition
Reliable delivery
Operations must commute and be idempotent
Operation-based CRDT

Convergence: sufficient condition

- Reliable delivery
  - Operations must commute and be idempotent

What if we have exactly-once delivery?

- We could drop idempotence requirement.
Operation-based CRDT

Convergence: sufficient condition
Reliable exactly-once delivery
Operations must commute and be idempotent
Operation-based CRDT

Convergence: sufficient condition

Reliable *exactly-once* delivery

Operations must commute and be idempotent

What if we have causal delivery?

Only concurrent operations need to commute.
Operation-based CRDT

Convergence: sufficient condition

Reliable causal exactly-once delivery

Concurrent Operations must commute and be idempotent
Example: highest timestamp wins register

State:

\( V_{ts} : \) value \( V \), timestamp \( ts \)

Operations:

\text{write}(V'_{ts'}) : \) if \( ts' > ts \) then \( V_{ts} := V'_{ts'} \)
Unified CRDT model

State-based:
- Payload type forms a semi-lattice
- Updates are increasing
- Merge computes Least Upper Bound
- Merge eventually delivered

Operation-based:
- Operations eventually delivered
- Concurrent operations commute

Combined:
- Operations are idempotent
CRDTs: Conflict-Free Replicated Data Types

Definition
Convergence conditions: State- vs. Op.-based

Concurrency semantics
Designing CRDTs
Beyond CRDTs
CRDTs: the unfortunate truth

Many data types do not naturally commute

Need to resolve concurrent operations
Set

Operations:
• \textit{add} (atom a)
• \textit{remove} (atom a)
• \textit{lookup} (atom a) : boolean

No duplicates
Extending the Set seq. spec.

Sequential specification of Set:
- \{true\} \text{add}(e) \quad \{e \in S\}
- \{true\} \text{rmv}(e) \quad \{e \not\in S\}

Commutative \((e \neq f)\):
- \{true\} \text{add}(e) ; \text{add}(e) \quad \{e \in S\}
- \{true\} \text{rmv}(e) ; \text{rmv}(e) \quad \{e \not\in S\}
- \{true\} \text{add}(e) ; \text{add}(f) \quad \{e, f \in S\}
- \{true\} \text{rmv}(e) ; \text{rmv}(f) \quad \{e, f \not\in S\}
- \{true\} \text{add}(e) ; \text{rmv}(f) \quad \{e \in S, f \not\in S\}

Ambiguous:
- \{true\} \text{add}(e) || \text{rmv}(e) \quad \{?????\}
Extending the Set seq. spec.

Sequential specification of Set:
- \{true\} \textbf{add}(e) \quad \{e \in S\}
- \{true\} \textbf{rmv}(e) \quad \{e \notin S\}

Commutative \((e \neq f)\):
- \{true\} \textbf{add}(e) ; \textbf{add}(e) \quad \{e \in S\}
- \{true\} \textbf{rmv}(e) ; \textbf{rmv}(e) \quad \{e \notin S\}
- \{true\} \textbf{add}(e) ; \textbf{add}(f) \quad \{e,f \in S\}
- \{true\} \textbf{rmv}(e) ; \textbf{rmv}(f) \quad \{e, f \notin S\}
- \{true\} \textbf{add}(e) ; \textbf{rmv}(f) \quad \{e \in S, f \notin S\}

What about:
- \{true\} \textbf{add}(e) \parallel \textbf{rmv}(e) \quad \{?????\}

“Principle of permutation equivalence”
Extending the Set seq. spec.

Sequential specification of Set:
- \( \{\text{true}\} \quad \text{add}(e) \quad \{e \in S\} \)
- \( \{\text{true}\} \quad \text{rmv}(e) \quad \{e \notin S\} \)

Commutative \((e \neq f)\):
- \( \{\text{true}\} \quad \text{add}(e) \parallel \text{add}(e) \quad \{e \in S\} \)
- \( \{\text{true}\} \quad \text{rmv}(e) \parallel \text{rmv}(e) \quad \{e \notin S\} \)
- \( \{\text{true}\} \quad \text{add}(e) \parallel \text{add}(f) \quad \{e, f \in S\} \)
- \( \{\text{true}\} \quad \text{rmv}(e) \parallel \text{rmv}(f) \quad \{e, f \notin S\} \)
- \( \{\text{true}\} \quad \text{add}(e) \parallel \text{rmv}(f) \quad \{e \in S, f \notin S\} \)

What about:
- \( \{\text{true}\} \quad \text{add}(e) \parallel \text{rmv}(e) \quad \{?????\} \)
add(e) || rem(e)

\{\textit{true}\} add(e) || rmv(e) \quad \{????\}

- linearisable? \_\_\_\_
- last writer wins? \{ add(e) < rmv(e) \Rightarrow e \notin S \wedge rmv(e) < add(e) \Rightarrow e \in S \}
- error state? \{ \bot e \in S \}
- add wins? \{ e \in S \}
- remove wins? \{ e \notin S \}

Deterministic
- Independent of order of delivery
- Independent of local state
- No synchronisation

[From strong to eventual consistency: getting it right]
Concurrent ≠ Sequential Interleaving

Consider Set-like object $S$ such that:

- $\{\text{true}\}$ add($e$) $\{e \in S\}$
- $\{\text{true}\}$ remove($e$) $\{e \not\in S\}$
- $\{\text{true}\}$ add($e$) $\|\$ remove($e$) $\{e \in S\}$

Satisfies correctness conditions:

| $\{\text{true}\}$ add($e$) $\|\$ remove($e'$) $\{e, e' \in S\}$ |
| $\{\text{true}\}$ add($e'$) $\|\$ remove($e$) $\{e, e' \in S\}$ |

Cannot be explained by any sequential interleaving

[From strong to eventual consistency: getting it right]
P-O Semantics of add-wins set

Set
On concurrent add/rem, add wins
P-O Semantics of add-wins set

Set
On concurrent add/rem, add wins
CRDTs: Conflict-Free Replicated Data Types

Definition
Convergence conditions: State- vs. Op.-based
Concurrency semantics
Designing CRDTs
Beyond CRDTs
Add-wins set (ORSet): operation-based based CRDT

**Problem**: how to guarantee that:

\[ \text{rem( UNL)} \ || \ add( \text{UNL}) = \{ \text{UNL} \} \]
Add-wins set (ORSet): operation-based CRDT

Key ideas:
(1) Tag information with **uids** (▲, Ỳ, □, ...)

\[ S = \{(UNL, ▲)\} \]

---

[From strong to eventual consistency: getting it right]
Add-wins set (ORSet): operation-based CRDT

Key ideas:
(1) Tag information with **uids** (▲, ■,...)

(2) Divide operations into:
   - **prepare**: with no side effect
   - **effect**: apply side-effects
Add-wins set (ORSet): operation-based CRDT

UID life-cycle
(1) non-existent
(2) created
(3) removed
(4) garbage-collected

Can we garbage-collect after remove operation?
Yes, causal delivery guarantees for any uid ◆:
\[ add(◆) \rightarrow rem(◆) \]
Operation-based OR-Set specification

**State:** set $S$ of pairs $(a, uid_a)$

$add(a) -> add( (a, uid_a))$

$S := S \cup \{(a, uid_a)\}$

$rem(a) -> rem( old_uids) : old_uids = \{u: (a, u) \in S\}$

$S := S \setminus old_uids$

$lookup() : returns \{a: (a, _) \in S\}$
Add-wins set (ORSet): state-based CRDT

**Problem**: what to do when the state of one replica includes u and the other does not?
**Add-wins set (ORSet): state-based CRDT**

**UID life-cycle**
1. non-existent
2. created
3. deleted
4. garbage-collected

Can we garbage-collect after remove operation?
No - need to keep tombstones.
Add-wins set (ORSet): state-based CRDT

**UID life-cycle**

1. non-existent
2. created
3. deleted
4. garbage-collected
Add-wins set (ORSet): state-based CRDT

**UID life-cycle**

1. non-existent
2. created
3. deleted
4. garbage-collected
State-based OR-Set specification

State, $S$
- $S.A = \text{set of pairs } (a, uid_a)$
- $S.R = \text{set of tombstone uids}$

add $(a) \rightarrow \text{add( (a, uid_a))}$
- $S.A := S \cup \{(a, uid_a)\}$

rem $(a) \rightarrow \text{rem( olduids) : olduids} = \{u : (a, u) \in S\}$
- $S.A := S.A \setminus \{(a, u) : u \in olduids\} \quad ; \quad S.R = S.R \cup olduids$

lookup() : returns $\{a : (a, _) \in S.A\}$
CRDTs: Conflict-Free Replicated Data Types

Definition
Convergence conditions: State- vs. Op.-based
Concurrency semantics

Designing CRDTs: Optimizing Metadata
Beyond CRDTs
Add-wins set (ORSet): state-based CRDT

Problem: can we add information to support garbage collecting removed uids?
Add-wins set (ORSWOT): state-based CRDT

**Key idea:** summarise removed information

- Tag element with logical timestamp
- Vector summarises adds received
- Merge: check vector: element is new?
- Preserves monotonic semi-lattice, $\equiv$ OR-Set
Add-wins set (ORSWOT): state-based CRDT

Key idea: summarise removed information

Tag element with logical timestamp
Vector summarises adds received
Merge: check vector: element is new?
Preserves monotonic semi-lattice, ≡ OR-Set

**u_{1.2} was removed because** \((1.2) \in [0,1]\)

**u_{1.1} was created because** \((1.1) \notin [0,1]\)
Library of CRDTs

<table>
<thead>
<tr>
<th>Register</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Last-Writer Wins</td>
<td>• Unlimited</td>
</tr>
<tr>
<td>• Multi-Value</td>
<td>• Non-negative</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Set</th>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Grow-Only</td>
<td>• Directed</td>
</tr>
<tr>
<td>• 2P</td>
<td>• Monotonic DAG</td>
</tr>
<tr>
<td>• Observed-Remove</td>
<td>• Edit graph</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Map</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>File-system tree</td>
<td></td>
</tr>
</tbody>
</table>
CRDTs: Conflict-Free Replicated Data Types

Definition

Convergence conditions: State- vs. Op.-based

Concurrency semantics

Designing CRDTs

Beyond CRDTs: Transactions & Invariants
Transactions

Snapshot isolation (conceptually)

**Begin**: take database snapshot

**Read/Write**: execute on database snapshot

**Commit**: success if no write/write conflict in concurrent transaction
Transactions

Parallel Snapshot isolation (conceptually)

- **Begin**: take database snapshot from a single replica
- **Read/Write**: execute on database snapshot
- **Commit**: success if no write/write conflict in concurrent transaction even on concurrent write/write in a CRDT

From: Y Sovran, et.al. Transactional storage for geo-replicated systems. In SOSP’11.
Parallel snapshot isolation: anomalies

T1: r(△) r(□) w(△)

T2: r(△) r(□) w(□)

T1 commit △, □

T2 commit △, □

Short fork
(allowed by snapshot isolation)

Long fork
(disallowed by snapshot isolation)

T1 commit △, □

T2 commit △, □

T1 and T2 can commit locally if it is known that no conflict exists

T1: r(△) r(□) w(△)

T3: r(△) r(□)

T2: r(△) r(□) w(□)

T4: r(△) r(□)

From: Y Sovran, et.al. Transactional storage for geo-replicated systems. In SOSP’11.
SwiftCloud: geo-replication right to the client machine

Extreme numbers of clients
Large database
Causal consistency for servers and clients
Fault tolerance

Switching servers?

Metadata proportional to # clients?

Full replication?
SwiftCloud: geo-replication right to the client machine

**Low latency**

... for **reads**

Partial replication/caching in clients

... for **writes**

Weak consistency

Mergeable writes => CRDT

+ Transactions with parallel snapshot isolation (PSI transactions)
Causal consistency in clients

**Problem**
Metadata of size $O(\#\text{clients})$ does not scale

**Key idea:**
Assign server timestamp when operation received in DC
Use version vector of size $O(\#\text{DCs})$ for tracking causal dependencies
Use client timestamp to filter duplicates
From strong to eventual consistency: getting it right

Transaction execution on server

sequencer [0 0 0] → (C,1) → (IE,1) → [1 0 0]

x
x_{000} = 8

y
y_{000} = 4

T(IE,1): add(2)

T(C,1) → (IE,1) : y.add(2)

commit T(C,1)

client
T(C,1): y.add(2)

surrogate

y_{000} = 4
Y_{100} = 6
From strong to eventual consistency: getting it right

Transaction execution on server

sequencer

x

y

IE (EU)

surrogate

client

T(C,1) ➔ (IE,1)

T(CA,1): sub(3)

T(IE,1): add(2)

Commit T(C,1)

T2.start dep = [0 0 0]

T: x.get() = 8
Support DC-failures or Client/DC network failure

Problem: on fail-over, new DC may not know some observed updates
  Leads to blocking or breaking session guarantees
Potential solutions
  Operations only complete after being stable in f+1 DCs
    Slow writes
  Involve clients in fail-over: clients keep enough information to ensure session guarantees
Client-assisted failover

Own updates
   Keep a log of own updates

Observable state
   Union of own updates and stable updates

On fail-over
   Replay log of own updates => may lead to double delivery
   Guarantee idempotency – rely on CRDT properties; use client identifiers in operation execution
CRDTs: Conflict-Free Replicated Data Types

Definition
Convergence conditions: State- vs. Op.-based
Concurrency semantics
Designing CRDTs
Beyond CRDTs: Transactions & Invariants
Limitations of PSI transactions

Similar to snapshot isolation
Not serializable
Cannot maintain (all) invariants across data items
E.g.: \textit{invariant}: \( \forall x, x \in \text{groups} \Rightarrow x \in \text{participants} \)
\text{participants}.rem("Nuno") || \text{groups}.add("Nuno","Marc")

Unlike snapshot isolation
Cannot maintain (some) invariants for a single data item
E.g.: \textit{invariant}: stock \geq 0
stock.dec(1) || stock.dec(1)
Invariant-preserving CRDTs (InvCRDT)

Strong consistency/\textit{serializability} is often used to guarantee that an \textit{invariant} is \textbf{not broken}

E.g.: $\textit{stock} \geq 0$

Could rely on \textbf{weak consistency} if \textit{invariant-preservation} is guaranteed
InvCRDT: e.g.: Non-negative counter

Key idea: local invariants that imply global invariants

Global invariant: $x \geq 0$

E.g. $x = 100$

$x$ can be decremented by 100

Split rights by the replicas

for 5 replicas, each replica can decrement $x$ by 20

each replica can additionally decrement $x$ by the value it incremented $x$

Operations that breaks local invariant needs to obtain additional rights

[P. O’Neil, The escrow transactional method, PODS’86]
When InvCRDTs are not enough...

...combine CRDTs with strong consistency

Walter [SOSP’11]

Gemini - red/blue consistency model [OSDI’12]
Road ahead

New data types
Composing CRDT
Dynamic sharing and slicing of CRDTs
Understand semantics (PLV community)
  E.g. several papers at POPL addressing weak consistency
Building effective EC databases with better guarantees
  Causal consistency
  Invariant-preservation
Transactions
Security
Final remarks

Principled approach
Two sufficient conditions
  State: monotonic semi-lattice
  Operation: commutativity
Useful CRDTs
  Register, Counter, Set, Map (KVS), Graph, Monotonic DAG, Sequence
Beyond simple CRDTs
  Transactions, invariant-preserving CRDT, etc.
CRDTs in the wild

Walter [SOSP 2011]: c-set
SwiftCloud: CRDTs + transactions + FT session guarantees

Riak currently supports counters; version 2.0 will support additional CRDTs (set, map)
Third-party implementations: StateBox (Erlang), KnockBox (Clojure), meangirls (Ruby), riak-java-crdt (Java)
Contributors

Annette Bieniusa
Carlos Baquero
Joan Marquès
Mahsa Najafzadeh
Marek Zawirski
Marc Shapiro
Mihai Leția
Nuno Preguiça
Sérgio Duarte
Valter Balegas
Strong consistency is easy to understand, but it requires synchronisation, which has a high cost and does not tolerate partial system failure or network partitions well. Hence, the past decade has seen renewed interest in an alternative - Eventual Consistency (EC) - where data is accessed and modified without coordination. Many production systems guarantee only EC, including NoSQL databases, key-value stores and cloud storage systems. EC improves responsiveness and availability, updates propagate more quickly, and enables less coordination-intensive and often less complex protocols.

EC is crucial for scalability but represents an unfamiliar design space. Conflicts and divergence are possible, making them hard to program against, while metadata growth and maintaining system-wide invariants can also become problematic. As more developers interact with and program against distributed storage systems, understanding and managing these trade-offs becomes increasingly important.

Although EC remains poorly understood, there has recently been considerable momentum in the research and development community: the past several years have seen the (re)introduction of useful concepts, such as replicated data types, monotonic programming, causal consistency, red-blue consistency, novel proof systems, etc., designed to allow improve efficiency, programmability, and overall operation of weakly consistent stores.

This workshop aims to investigate the principles and practice of Eventual Consistency. It will bring together theoreticians and practitioners from different horizons: system development, distributed algorithms, concurrency, fault tolerance, databases, language and verification, including both academic and industry. In order to make EC computing easier and more accessible, and to address the limitations of EC and explore the design space spectrum between EC and strong consistency, we will share experience with practical systems and from theoretical study, to investigate algorithms, design patterns, correctness conditions, complexity results, and programming methodologies and tools.