Data Replication and Consistency

Data Consistency

Operational Transformation Approaches

Conflict-free Replicated Data Structures

Gérald Oster,
Associate Professor, TELECOM Nancy, Université de Lorraine
COAST Team, LORIA, Université de Lorraine, Inria, CNRS
**CAP Theorem** [GL02]

- A system without network partition, can achieve consistency + availability
  - Client and storage system are part of the same environment
- In a large-scale distributed systems, network partitions exist
  - Consistency and availability cannot be both achieved
    - Relax consistency, maintain availability
    - Maintain consistency, tolerate unavailability under certain conditions
Optimistic Replication

• To sidestep CAP theorem [GL02], a first solution is to avoid synchronizing replicas. Replicas simply reconcile their copies in the background.

• Trade-off between consistency and availability
  • allows replicas to diverge
• This approach is named lazy [LLS90], or optimistic replication [SS05]

• Examples: Bayou, Amazon S3, etc.
Consistency in Distributed Systems

Eventual Consistency

- **Definition** *(eventual consistency)*
  
  A history $h$ is eventually consistent (EC) when for every object $x$ if there is a bounded amount of write operations on $x$ in $h$, then eventually all the read operation observer the same state.

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td></td>
</tr>
<tr>
<td>w(x₁)</td>
<td>r(x₁)</td>
</tr>
<tr>
<td>q</td>
<td></td>
</tr>
<tr>
<td>w(x₂)</td>
<td>r(x₂)</td>
</tr>
</tbody>
</table>
```
Optimistic Replication

Strong Eventual Consistency

• Strong Eventual Consistency
  • Eventual delivery: « An update executed at some correct replica eventually executes at all correct replicas »
  • Strong convergence = correct replicas that have executed the same updates have equivalent state
  • No consensus in background, no need to rollback
Operational Transformation (OT)
Operation-based approach model

- \( n \) copies of an object hosted at \( n \) sites
- An object is modified by applying operations

- Each operation is
  - generated at a site (local execution),
    and applied immediately on the local copy
  - broadcasted to other sites
  - integrated at those sites (remote execution)

- System is correct if when it is idle all copies are identical (SEC)
Operational Transformation [EG89]

General Architecture

- 2 components:
  - An integration algorithm (diffusion, integration)
  - A set of transformation functions (conflict resolution)

- 3 main issues:
  - Convergence (~ EC)
  - Causality violation
  - (Intention violation)
Operational Transformation
Running Example

- Textual document = sequence of characters
- Operations:
  - $\text{Ins}(p, c)$ – inserts character 'c' at position 'p'
  - $\text{Del}(p)$ – removes character at position 'p'

\[
\begin{align*}
\text{Site 1} & \quad \text{concurrency control} \\
\text{op}_1 &= \text{ins}(7, r) \\
\text{op}_2 &= \text{ins}(17, o) \\
\end{align*}
\]

\[
\begin{align*}
\text{Site 2} & \quad \text{concurrency control} \\
\text{op}_2 &= \text{ins}(17, o) \\
\text{op}'_2 &= \text{ins}(18, o) \\
\end{align*}
\]

\[
\begin{align*}
T(\text{Ins}(p_1, c_1), \text{Ins}(p_2, c_2)) :& \quad \text{if } (p_1 < p_2) \text{ return } \text{Ins}(p_1, c_1) \\
& \quad \text{else return } \text{Ins}(p_1 + 1, c_1) \\
\end{align*}
\]
Operational Transformation
Example of Transformation Functions

\[ T(\text{Ins}(p1,c1), \text{Ins}(p2,c2)) \] :
\[
\begin{align*}
& \text{if } (p1 < p2) \text{ return } \text{Ins}(p1,c1) \\
& \text{else return } \text{Ins}(p1+1,c1)
\end{align*}
\]

\[ T(\text{Ins}(p1,c1), \text{Del}(p2)) \] :
\[
\begin{align*}
& \text{if } (p1 \leq p2) \text{ return } \text{Ins}(p1,c1) \\
& \text{else return } \text{Ins}(p1-1,c1) \\
& \text{endif}
\end{align*}
\]

\[ T(\text{Del}(p1), \text{Ins}(p2,c2)) \] :
\[
\begin{align*}
& \text{if } (p1 < p2) \text{ return } \text{Del}(p1) \\
& \text{else return } \text{Del}(p1+1)
\end{align*}
\]

\[ T(\text{Del}(p1), \text{Del}(p2)) \] :
\[
\begin{align*}
& \text{if } (p1 < p2) \text{ return } \text{Del}(p1) \\
& \text{else if } (p1 > p2) \text{ return } \text{Del}(p1-1) \\
& \text{else return } \text{Id}()
\end{align*}
\]
Operational Transformation
Correctness [EG89]

(TP1) \( op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2) \)

- \( op_1 \) and \( op_2 \) concurrent, defined on a state \( S \)
- \( op'_2 \) same effects as \( op_2 \), defined on \( S \cdot op_1 \)
Operational Transformation Correctness [RNG96]

(TP2) \[ T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2)) \]
Operational Transformation
Causality Violation

\[ op_1 = \text{ins}(1, y) \]

\[ op' = \text{del}(1) \]

\[ op_2 = \text{del}(1) \]

\[ op_1 = \text{ins}(1, y) \]

\[ op_2 = \text{del}(1) \]

\[ op_1 = \text{ins}(1, y) \]
Operational Transformation
Causality Preservation

Site 1

Site 2

Site 3

\[ \text{op}_1 = \text{ins}(1, y) \]

\[ \text{op}'_2 = \text{del}(1) \]

\[ \text{op}_2 = \text{del}(1) \]

\[ \text{op}_1 = \text{ins}(1, y) \]

\[ \text{op}_1 = \text{ins}(1, y) \]

\[ \text{op}_2 = \text{del}(1) \]

\[ \text{op}_2 = \text{del}(1) \]

\[ \text{op}_1 = \text{ins}(1, y) \]

\[ \text{op}_2 = \text{del}(1) \]
Operational Transformation (OT)
Operation Intention

• Intention of an operation is the observed effect as result of its execution on its generation state

• Passing from an initial state ‘ab’ to a final state ‘aXb’ could be observed as:
  • ins(2,x)
  • ins(a < X < b)
  • ins(a < X)
  • ins(X < b)
Operational Transformation (OT)

Intention Preservation [SJZYC98]

• For any operation \( op \) the effects of executing \( op \) at all sites should be the same as the intention of \( op \).

• The effect of executing \( op \) does not change the effects of independent operations.
Operational Transformation (OT)

Existing Approaches

- Two main families:
  - Transformation functions satisfying both TP1 and TP2
  - Control algorithms avoiding (needs of) TP2
Operational Transformation (OT)  
Existing Approaches

• Design and verify transformation functions T
• T also known as forward transposition (\textit{transpose}_{fd})
• Verification of conditions TP1 and TP2
  • Combinatorial explosion (>100 cases for a string)
  • Iterative process
  • Repetitive and error prone task
Operational Transformation (OT)
Partial Concurrency

Site 1

\[ \text{op}_1 = \text{ins}(5,p) \]

\[ \text{op}_3 = \text{ins}(6,h) \]

\[ \text{op''}_2 = \text{del}(7) \]

Site 2

\[ \text{op}_2 = \text{del}(5) \]

\[ \text{op'}_1 = \text{ins}(5,p) \]

\[ \text{op'}_2 = T(\text{op}_2, \text{op}_1) = \text{del}(6) \]

\[ \text{op''}_2 = T(\text{op'}_2, \text{op}_3) = \text{del}(7) \]

\[ \text{op'}_1 = T(\text{op}_1, \text{op}_2) = \text{ins}(5) \]

\[ T(\text{op}_3, \text{op}_2) \]

is not allowed to be performed

Nor \( T(\text{op}_3, \text{op'}_1) \)
Operational Transformation (OT)  
Partial Concurrency
Operational Transformation (OT) Backward Transposition [SJZYC98]

• Backward transposition (\(\text{transpose}_bk\))

\[
\text{transpose}_bk(\text{op}_1, \text{op’}_2) = (\text{op}_2, \text{op’}_1)
\]

• \(\text{op’}_2 = T(\text{op}_2, \text{op}_1)\)
• \(\text{op’}_1 = T(\text{op}_1, \text{op}_2)\)

• Note:
  • \(\text{transpose}_bk(\text{op}_a, \text{op}_b) = (T^{-1}(\text{op}_b, \text{op}_a), T(\text{op}_a, T^{-1}(\text{op}_b, \text{op}_a)))\)
Operational Transformation

Example of Transformation Functions [RNG96]

\[ T(\text{Ins}(p_1,c_1,u_1), \text{Ins}(p_2,c_2,u_2)) : \]
  \[ \text{if } (p_1 < p_2) \text{ or } (p_1 = p_2 \text{ and } u_1 < u_2) \text{ return } \text{Ins}(p_1,c_1,u_1) \]
  \[ \text{else return } \text{Ins}(p_1+1,c_1,u_1) \]

\[ T(\text{Ins}(p_1,c_1,u_1), \text{Del}(p_2,u_2)) : \]
  \[ \text{if } (p_1 \leq p_2) \text{ return } \text{Ins}(p_1,c_1,u_1) \]
  \[ \text{else return } \text{Ins}(p_1-1,c_1,u_1) \]
  \[ \text{endif} \]

\[ T(\text{Del}(p_1,u_1), \text{Ins}(p_2,c_2,u_2)) : \]
  \[ \text{if } (p_1 < p_2) \text{ return } \text{Del}(p_1,u_1) \]
  \[ \text{else return } \text{Del}(p_1+1,u_1) \]

\[ T(\text{Del}(p_1,u_1), \text{Del}(p_2,u_2)) : \]
  \[ \text{if } (p_1 < p_2) \text{ return } \text{Del}(p_1,u_1) \]
  \[ \text{else if } (p_1 > p_2) \text{ return } \text{Del}(p_1-1,u_1) \]
  \[ \text{else return } \text{Id}() \]

Satisfy TP1 but not TP2!
Operational Transformation
Example of Violation of TP2

User 1

abc

ac

ayc

User 2

abc

abye

ayc

User 3

abc

axye

\[ \text{op}_2 = \text{del}(2, b) \]

\[ \text{op}_3 = \text{ins}(3, y) \]

\[ \text{op'}_2 = \text{ins}(2, y) \]

\[ \text{op'}_3 = \text{del}(2) \]

\[ \text{op'}_1 = \text{ins}(3, x) \]

\[ \text{op''}_1 = \text{ins}(2, x) \]
Operational Transformation (OT)
False-Tie Problem

Site 1
- abc
- axbc

Site 2
- abc
- ac

Site 3
- abc

$op_1 = ins(2, x)$
$op_2 = del(2, b)$
$op_3 = ins(3, y)$

$op'_1 = ins(2, x)$
$op'_3 = ins(2, y)$

$axyc \ ?$

or

$ayx \ ?$

or

$ayxc \ ?$
Operational Transformation (OT)
Tombstone Transformation Functions \[OUMI06]\n
- Keep 'tombstones' of deleted elements
Operational Transformation (OT)

Tombstone Transformation Functions

\[ T(\text{Ins}(p_1, c_1, s_1), \text{Ins}(p_2, c_2, s_2)) : \]
- \( \text{if } (p_1 < p_2) \text{ or } (p_1 = p_2 \text{ and } s_1 < s_2) \text{ return } \text{Ins}(p_1, c_1, s_1) \)
- \( \text{else return } \text{Ins}(p_1 + 1, c_1, s_1) \)

\[ T(\text{Ins}(p_1, c_1, s_1), \text{Del}(p_2, s_2)) : \]
- \( \text{return } \text{Ins}(p_1, c_1, s_1) \)

\[ T(\text{Del}(p_1, s_1), \text{Ins}(p_2, c_2, s_2)) : \]
- \( \text{if } (p_1 < p_2) \text{ return } \text{Del}(p_1, s_1) \)
- \( \text{else return } \text{Del}(p_1 + 1, s_1) \)

\[ T(\text{Del}(p_1, s_1), \text{Del}(p_2, s_2)) : \]
- \( \text{return } \text{Del}(p_1, s_1) \)
Operational Transformation (OT)

Tombstone Transformation Functions – Compact Model

Compacted model = sequence of (character, absolute-position)

View

a b y c

Model

h a b r n y c

Insert(3,y)

view2model()

Insert(5,y)

Compacted

a,2 b,3 y,5 c,6

Insert(3,y)

view2model()

Insert(5,y)
Operational Transformation (OT)

Tombstone Transformation Functions – Delta Model

**Delta model** = sequence of (character, offset)

View

```
View
a  b  y  c
```

Compressed

```
Compressed
a,2  b,3  y,5  c,6
```

```python
view2model()
```

Delta

```
Delta
a,+2  b,+1  y,+2  c,+1
```

```python
Insert(3,y)
view2model()
```

```python
Insert(5,y)
```
Operational Transformation (OT)
Tombstone Transformation Functions – Comparison

• Basic Model
  • Deleted characters are kept
  • Size of the model is growing infinitely

• Compacted Model
  • Update absolute position of all characters located after the effect position

• Delta Model
  • Update the offset of next character

• Observations
  • view2model can be optimized (caret position)
  • Overhead of view2model is not significant
Operational Transformation (OT)
General Control Algorithm – SOCT2 [SCF97]

a) The initial history buffer

\[ \text{HB} \quad \text{op}_1 \quad \text{op}_2 \quad \text{op}_3 \quad \ldots \quad \text{op}_i \quad \ldots \quad \text{op}_j \quad \ldots \quad \text{op}_n \]

- remote operation
- the operation precedes the remote operation
- the operation is concurrent with the remote operation

b) principle of integration

equivalent HB

- operations preceding the remote operation
- operations concurrent with the remote operation

forward transposition

\[ \text{op}_{n+1} \]
Operational Transformation (OT)
Control Algorithms that avoid TP1 and/or TP2

• GOT algorithm [SJZYC98]
  • Does not need to satisfy TP1 and TP2
  • Requires a global serialization order
    • Sum of state vector components
    • If equality, then priority on sites
  • Requires very costly undo/redo mechanism
    • Does not ensure SEC (only EC)
Operational Transformation (OT)
Control Algorithms that avoid TP1 and/or TP2

• SOCT4 Algorithm [VCFS00]
  • No undo/redo mechanism, no state vectors
  • Eliminates TP2, but requires TP1
  • Global order of operations according to timestamps generated by a sequencer
  • Local operations executed immediately
  • Assigns a timestamp to the operation and transmits it to the other sites
  • Defers broadcast until execution of all preceding operations
  • Transformations performed by each site
Operational Transformation (OT)
SOCT4 Algorithm [VCFS00]
Operational Transformation (OT)  
SO6 [MOSI03]– SOCT4 Variant

\[
\text{Sync}(\log, N_s) : \quad \text{while } ((\text{op}_r = \text{getOp}(N_s+1))\neq \emptyset) \quad  \\
\text{for } (i = 0; i < \log.\text{size}(); i++) \quad  \\
\quad \text{op}_i = \log[i]; \quad \text{log}[i] = T(\text{op}_i, \text{op}_i) \quad \text{op'}_i = T(\text{op}_i, \text{op}_i); \quad \text{endif} \quad \text{execute}(\text{op'}_i) \quad \text{endfor} \quad N_s = N_s + 1 \quad \text{endwhile} \quad  \\
\text{for } (i = 0; i < \log.\text{size}(); i++) \quad  \\
\quad \text{op'}_i = \log[i]; \quad  \\
\quad \text{if } \text{send}(\text{op'}_i, N_s+1) \text{ then} \quad \text{endif} \quad \\
\quad N_s = N_s + 1 \quad \text{else} \quad \text{error} \quad \text{need to synchronize} \quad \text{endif} \quad \text{endfor}
\]

\text{getOp(ticket)} \text{ retrieves operation identified by timestamp ticket}

\text{send(op,ticket)} \text{ sends local operation with timestamp ticket. If ticket already exists, returns false}
Operational Transformation (OT)
SO6 [MOSI03]– SOCT4 Variant

Site 1, Ns=0
- op_1
- op_2
- s_1=synchronize()

Site 2, Ns=0
- op_3
- op_4
- s_2=synchronize()

**At step s_1**
- sync([op_1,op_2],0)
- Send op_1, op_2
- Ns=2

**At step s_2**
- sync([op_3,op_4],0)
- op'_1=T(op_1,op_3)
- op'_3=T(op_3,op_1)
- op''_1=T(op'_1,op_4)
- op'_4=T(op_4,op'_1)
- op'_2=T(op_2,op'_3)
- op''_3=T(op'_3,op_2)
- op''_2=T(op'_2,op'_4)
- op''_4=T(op'_4,op'_2)
- op''_1, op''_2 are executed
- op''_3, op''_4 are sent
- Ns=4

**At step s_3**
- sync([],2)
- op''_3 and op''_4 are executed
- Ns=4
Operational Transformation (OT)
SO6 [MOSI03]– SOCT4 Variant

• At site 1
  • $\text{op}_1$
  • $\text{op}_2$
  • $\text{op''}_3 = T(\text{op}_3, \text{op}_1), \text{op}_2$
  • $\text{op''}_4 = T(\text{op}_4, \text{op'}_1), \text{op'}_2$

• At site 2
  • $\text{op}_3$
  • $\text{op}_4$
  • $\text{op''}_1 = T(\text{op}_1, \text{op}_3), \text{op}_4$
  • $\text{op''}_2 = T(\text{op}_2, \text{op'}_3), \text{op'}_4$
Operational Transformation (OT)
Control Algorithms that avoid TP1 and/or TP2

• Jupiter algorithm [NCDL95]
  • Used in GoogleDocs
  • Requires a central server
  • Eliminates TP2, but requires TP1
  • Does not need state vectors
  • Transformations done on the server + client side
Operational Transformation (OT)
Jupiter algorithm [NCDL95]

- GoogleWave

... now in GoogleDocs
Operational Transformation (OT)

Jupiter Algorithm

\[\text{xform}(c,s) = \{c',s'\}\]

\[\text{xform}(\text{del } x, \text{del } y) =\]

\[\begin{cases} 
\{\text{del } x-1, \text{del } y\} & \text{if } x>y \\
\{\text{del } x, \text{del } y-1\} & \text{if } x<y \\
\{\text{no-op, no-op}\} & \text{if } x=y 
\end{cases}\]
Operational Transformation (OT)

Jupiter Algorithm

(a)

(b)
Operational Transformation (OT)

Jupiter Algorithm
Operational Transformation (OT)

Jupiter Algorithm

```c
int myMsgs = 0; /* number of messages generated */
int otherMsgs = 0; /* number of messages received */

generate(op) {
    apply op locally;
    send(op, myMsgs, otherMsgs);
    add(op, myMsgs) to outgoing;
    myMsgs = myMsgs + 1;
}
receive(msg) {
    /* discard acknowledged messages */
    for m in (outgoing) {
        if (m.myMsgs < msg.otherMsgs)
            remove m from outgoing;
    }
    /* ASSERT: msg.myMsgs == otherMsgs */
    for i in [1..length(outgoing)] {
        /* transform new messages and the ones in the queue */
        { msg, outgoing[i] } = xform(msg, outgoing[i]);
    }
    apply msg.op locally;
    otherMsgs = otherMsgs + 1;
}
```
\( \text{op}_1 = T(\text{op}_3, \text{op}_1) \)

\( \text{op}_2 = T(\text{op}_3, \text{op}_1) \)

\( \text{op}_3 = T(T(\text{op}_3, \text{op}_1), \text{op}_2) \)

\( \text{op}_4 = T(T(\text{op}_2, T(\text{op}_3, \text{op}_1)), \text{op}_1) \)

\( \text{op'}_3 = T(T(\text{op}_3, \text{op}_1), \text{op}_2) \)

\( \text{op'}_4 = T(T(\text{op}_2, T(\text{op}_3, \text{op}_1)), \text{op}_1) \)

\( \text{outgoing} = \{ \text{op}_1 \} \)

\( \text{outgoing} = \{ \text{op}_1, \text{op}_2 \} \)

\( \text{outgoing} = \{ T(\text{op}_1, \text{op}_3), T(\text{op}_2, T(\text{op}_3, \text{op}_1)) \} \)

\( \text{outgoing} = \{ T(T(\text{op}_2, T(\text{op}_3, \text{op}_1)), \text{op}_4) \} \)

\( \text{outgoing} = \{ T(T(\text{op}_2, T(\text{op}_3, \text{op}_1)), \text{op}_4), \text{op}_4, T(\text{op}_2, T(\text{op}_3, \text{op}_1)) \} \)
Operational Transformation (OT)

Jupiter Algorithm – Generalization to n clients [Z01]

```plaintext
apply msg.op locally;

for (c in client list) {
    if (c != client) {
        send(c, msg);
    }
}

apply msg.op locally;
```

Algorithm changes at server side
Operational Transformation (OT)
Jupiter Algorithm – Summary

- Requires a server that performs transformations
- Not suitable for P2P environments
- False tie scenario gives different results according to integration order

\[
\begin{align*}
\text{Site 1} & : \text{abc} \\
\text{Site 2} & : \text{abc} \\
\text{Site 3} & : \text{abc}
\end{align*}
\]

- \(\text{op}_1 = \text{ins}(2, x)\)
- \(\text{op}_2 = \text{del}(2, b)\)
- \(\text{op}_3 = \text{ins}(3, y)\)

- \(\text{op}'_1 = \text{ins}(2, x)\)
- \(\text{op}'_3 = \text{ins}(2, y)\)

\[
\begin{align*}
\text{axbc} & \quad \text{abc} \\
\text{axyc} & \quad \text{abyce}
\end{align*}
\]
Operational Transformation (OT) Summary

- Transforms non commuting operations to make them commute
- Genericity
- Time complexity
  - Average: $O(H \cdot c)$
  - Worst case: $O(H^2)$
- Difficult to write correct transformation functions
- State vectors used for detecting concurrency ⇒ scalability limitations
- Not very suitable for large scale peer-to-peer collaboration
Conflict-free Replicated Data Types (CRDT) [SPBZ09]

- Data structures
  - Same semantics (w/o concurrency) as classical data-structures
  - Designs operations to be commutative
- Replicated
  - At multiple sites
- Available
  - Replica is updated without coordination
  - Convergence is (formally) guarantee
  - Decentralized / Peer-to-peer
- State-based and Operation-based approaches
Conflict-free Replicated Data Types
State-based Replication

- Replicated object: a tuple \((S, s_0, q, u, m)\)
  - \(S\): state domain
  - Replica at process \(p_i\) has state \(s_i \in S\)
  - \(s_0\): initial state
- Each replica can execute one of following commands
  - \(q\): query object’s state
  - \(u\): update object’s state
  - \(m\): merge state from a remote replica
Conflict-free Replicated Data Types
State-based Replication

- Algorithm
  - Periodically, replica at $p_i$ sends its current state to $p_j$
  - Replica $p_j$ merges received state into its local state by executing $m$
- After receiving all updates (irrespective of order), each replica will have same state
Conflict-free Replicated Data Types
Semi-lattice

• Partial order \( \leq \) set \( S \) with a least upper bound (LUB), denoted \( \sqcup \)
  • \( m = x \sqcup y \) is a LUB of \( \{ x, y \} \) under \( \leq \) if and only if
    \[ \forall m', x \leq m' \land y \leq m' \Rightarrow x \leq m \land y \leq m \land m \leq m' \]

• It follows that \( \sqcup \) is:
  • commutative: \( x \sqcup y = y \sqcup x \)
  • idempotent: \( x \sqcup x = x \)
  • associative: \( (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z) \)
Conflict-free Replicated Data Types

Semi-lattice – Example on integers

• Partial order $\leq$ on set of integers
• Least upper bound $\sqcup : \text{max}$ (maximum function)

• Therefore, we have:
  • commutative: $\text{max}(x, y) = \text{max}(y, x)$
  • idempotent: $\text{max}(x, x) = x$
  • associative: $\text{max}(\text{max}(x, y), z) = \text{max}(x, \text{max}(y, z))$
Conflict-free Replicated Data Types
Semi-lattice – Example on sets

• Partial order \( \subseteq \) on sets
• Least upper bound \( \sqcup \) : \( \cup \) (set union)

• Therefore, we have:
  • commutative: \( A \cup B = B \cup A \)
  • idempotent: \( A \cup A = A \)
  • associative: \( (A \cup B) \cup C = A \cup (B \cup C) \)
Conflict-free Replicated Data Types
Monotonic Semi-lattice Object

• A state-based object with partial order $\leq$, noted $(S, \leq, s_0, q, u, m)$, that has following properties, is called a monotonic semi-lattice:
  1. Set $S$ of values forms a semi-lattice ordered by $\leq$
  2. Merging state $s$ with remote state $s'$ computes the LUB of the two states, i.e., $s \cdot m(s') = s \sqcup s'$
     \textbf{(delivery order is not important)}
  3. State is monotonically non-decreasing across updates, i.e., $s \leq s \cdot u$
     \textbf{(updates have effect, no rollback)}
Conflict-free Replicated Data Types
Convergent Replicated Data Type (CvRDT)

• **Theorem:** Assuming eventual delivery and termination, any replicated state-based object that satisfies the monotonic semi-lattice property is SEC

• **Since:**
  - Merge is both **commutative** and **associative**
    • We do not care about order
  - Merge is **idempotent**
    • We do not care about delivering more than once
Convergent Replicated Data Types

Example

- Each replica can execute one of following commands
  - query q: returns entire set
  - update u: adds new element \((e, a)\) to local set
  - merge m: compute unions between local set and remote set
Conflict-free Replicated Data Types
Operation-based Replication

- Replicated object: a tuple \((S, s_0, q, t, u, P)\).
  - \(S\): state domain
  - Replica at process \(p_i\) has state \(s_i \in S\)
  - \(s_0\): initial state
- Each replica can execute one of following commands
  - \(q\): query object’s state
  - \(t\): side-effect-free prepare-update method (at local replica)
  - \(u\): effect-free update method (at all replicas)
  - \(P\): delivery precondition
Conflict-free Replicated Data Types
Operation-based Replication

- **Algorithm**
  - Updates are delivered to all replicas
  - Use causally-ordered broadcast communication protocol, i.e., deliver every message to every node exactly once, w.r.t. happen-before order
  - Happen-before: updates from same replica are delivered in the order they happened to all recipients (effectively delivery precondition, $P$)
  - Note: concurrent updates can be delivered in any order
Conflict-free Replicated Data Types
Commutativity Property

• Updates \((t, u)\) and \((t', u')\) commute, if and only if for any reachable replica state \(s\) where both \(u\) and \(u'\) are enabled:
  • \(u\) (resp. \(u'\)) remains enabled in state \(s \cdot u'\) (resp. \(s \cdot u\))
  • \(s \cdot u \cdot u' \equiv s \cdot u' \cdot u\)

• Commutativity holds for concurrent updates
Conflict-free Replicated Data Types
Commutative Replicated Data Type (CmRDT)

• **Theorem:** Assuming causal delivery of updates and method termination, any replicated op-based object that satisfies the commutativity property for all concurrent updates is SEC.
Commutative Replicated Data Types

Example

- query q: returns entire set
- prepare method t: adds new element \((e, \alpha)\) to local set
- update u: add delta to any remote replica

\[
\begin{align*}
\{5\} & \cup \{3\} = \{3, 5\} \\
\{3, 5\} & \cup \{7\} = \{3, 5, 7\} \\
\{5\} & \cup \{7\} = \{5, 7\} \\
\{5, 7\} & \cup \{3\} = \{3, 5, 7\}
\end{align*}
\]
Consistency Maintenance
Conflict-free Replicated Data Types (CRDT)

• Register
  • Last-Writer Wins
  • Multi-Value

• Set
  • Grow-Only
  • 2-Phase
  • Observed-Remove
  • Observed-Update-Remove

• Map
• Counter
• Graph
  • Directed
  • Monotonic DAG
  • Edit graph

• Sequence
**Conflict-free Replicated Data Types**

**Observed-Remove Set (CvRDT)**

- **Payload:** $(A, R)$ - added/removed sets of $(element, unique-token)$
- **Operations:**
  
  \[
  \begin{align*}
  add(e): & A := A \cup \{(e, a)\} \\
  remove(e): & R := R \cup \{(e, -) \in A\} \quad \text{remove all unique elements observed} \\
  lookup(e): & \exists (e, -) \in A \setminus R \\
  merge(S, S'): & (A \cup A', R \cup R')
  \end{align*}
  \]

**Optimised OR-set**

- summarise received adds in a vector
- check vector during merge
Conflict-free Replicated Data Types
Observed-Remove Set (CmRDT)

- Payload:
  \( S = \{(e, \alpha), (e, \beta), (e, \chi), \ldots\} \) where \( \alpha, \beta, \chi, \ldots \) are unique token

- Operations:
  \( \text{add}(e): \quad S := S \cup \{(e, \alpha)\} \) where \( \alpha \) is a fresh unique token
  \( \text{lookup}(e): \quad \exists \alpha: (e, \alpha) \in S \)
  \( \text{remove}(e): \quad R := \{(e, \alpha) | \exists \alpha (e, \alpha) \in S\} \) (at source) no tombstones
  \( S := S \setminus R \)
  \{\text{true}\} \quad \text{add}(e) \parallel \text{remove}(e) \quad \{e \in S\} \)
Conflict-free Replicated Data Types
P-Counter (CvRDT)

• Payload:
  • \( P = [\text{int, int, ...}] \)

• Operations:
  • \text{value}(): \( \sum_i P[i] \)
  • \text{increment}(): \( P[\text{MyID}]++ \)
  • \text{merge}(S,S'):: S \sqcup S' = [..., \max(s.P[i],s'.P[i]),...],i

• Positive
Conflict-free Replicated Data Types
PN-Counter (CvRDT)

• Payload:
  • $P = [\text{int}, \text{int}, ...]$
  • $N = [\text{int}, \text{int}, ...]$

• Operations:
  • `value()`: $\sum_i P[i] - \sum_i N[i]$
  • `increment()`: $P[\text{MyID}]++$
  • `decrement()`: $N[\text{MyID}]++$
  • `merge(S,S')`: $S \sqcup S' = ([..., \max(s.P[i],s'.P[i]), ...], i,$
    $[..., \max(s.N[i],s'.N[i]), ...])$

• Positive or negative
Conflict-free Replicated Data Types
CvRDT vs. CmRDT

- Both approaches are equivalent
  - A state-based object can emulate an operation-based object, and vice-versa

- Operation-based:
  - More efficient since you only ship small updates
  - But require causally-ordered broadcast

- State-based:
  - Only require reliable broadcast
Conflict-free Replicated Data Types (CRDT) (Text) Sequence [PMSL09] [WUM09]

- Document = linear sequence of elements
  - Each element has a unique identifier
  - Identifier constant for the lifetime of the document
  - Dense total order of identifiers consistent with element order:
    - \( \forall \text{id}_x, \text{id}_y: \text{id}_x < \text{id}_y \Rightarrow \exists \text{id}_z: \text{id}_x < \text{id}_z < \text{id}_y \)

- Different approaches for generating identifiers:
  - TreeDoc, Logoot, LogootSplit, …
Conflict-free Replicated Data Types (CRDT) 

Logoot [WUM09]

Logoot identifiers: \(<p_1,s_1,h_1><p_2,s_2,h_2> \ldots <p_k,s_k,h_k>\)

- \(p_i\) integer
- \(s_i\) site identifier
- \(h_i\) logical clock at site \(s_i\)

\[
\text{ins}(<3,2,5><13,1,7>, r)
\]

\[
\text{ins}(<12,3,1><7,8,2><13,3,6><7,2,9>, o)
\]

- Time complexity
  - Average: \(O(k \cdot \log(n))\)
  - Worst case: \(O(H \cdot \log(H))\)
- No need for concurrency detection
- Identifiers storage cost
- New design for each data type
- Suitable for large-scale collaboration
**Conflict-free Replicated Data Types (CRDT)**

**LogootSplit [AMOI13]**

LogootSplit identifiers

- **Base**
- **Interval**

| p₁ | ... | pₙ | site_id | clock | begin | end |

**Base** identifiers:

- 1,1,[0,16] concurrency control

**Interval** identifiers:

1,1,[0,5] concur
1,1,5,2,1,[0,0] r
1,1,[6,16] ency contr

- **Insert r** between “concur” and “ency contr”

- **Insert o** between “ency contr” and “l”

**Base** updated:

<table>
<thead>
<tr>
<th>1,1,[0,5] concur</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,1,5,2,1,[0,0] r</td>
</tr>
<tr>
<td>1,1,[6,15] ency contr</td>
</tr>
<tr>
<td>1,1,15,3,1,[0,0] o</td>
</tr>
<tr>
<td>1,1,[16,16] 1</td>
</tr>
</tbody>
</table>
Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]

Site 3

ABCDEF
2,1,[0,5]

insert XY between B and C

Site 4

ABCDEF
2,1,[0,5]
Conflict-free Replicated Data Types (CRDT)

**LogootSplit [AMOI13]**

**Site 3**

- ABCDEF
  - 2,1,[0,5]

- AB
  - 2,1,[0,1]

- XY
  - 2,1,[0,1]

- CDEF
  - 2,1,[2,5]

**Site 4**

- ABCDEF
  - 2,1,[0,5]

- AB
  - 2,1,[0,5]

- XYZ
  - 2,1,[0,1]

- CDEF
  - 2,1,[2,5]

**insert XY between B and C**
Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]

Site 3

ABCDEF 2,1,[0,5]

insert XY between B and C

AB 2,1,[0,1]  XY 2,1,13,1,[0,1]  CDEF 2,1,[2,5]

Site 4

ABCDEF 2,1,[0,5]

insert ZT between D and E

ABCD 2,1,[0,3]  EF 2,1,[4,5]
Conflict-free Replicated Data Types (CRDT)

LogootSplit [AMOI13]

Site 3

<table>
<thead>
<tr>
<th>AB</th>
<th>2,1, [0,1]</th>
<th>XY</th>
<th>2,1, [0,1]</th>
<th>CDEF</th>
<th>2,1, [2,5]</th>
</tr>
</thead>
</table>

insert XY between B and C

Site 4

<table>
<thead>
<tr>
<th>ABCD</th>
<th>2,1, [0,3]</th>
<th>ZT</th>
<th>2,1, [3,4,1], [0,1]</th>
<th>EF</th>
<th>2,1, [4,5]</th>
</tr>
</thead>
</table>

insert ZT between D and E

sending of operations

<table>
<thead>
<tr>
<th>AB</th>
<th>2,1, [0,1]</th>
<th>XY</th>
<th>2,1, 1,3,1, [0,1]</th>
<th>CDEF</th>
<th>2,1, [2,5]</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ABCD</th>
<th>2,1, [0,3]</th>
<th>ZT</th>
<th>2,1, 3,4,1, [0,1]</th>
<th>EF</th>
<th>2,1, [4,5]</th>
</tr>
</thead>
</table>
Conflict-free Replicated Data Types (CRDT)

**LogootSplit** [AMOI13]

Site 3

- **ABCDEF**
  - 2,1,[0,5]

- **AB**
  - 2,1,[0,1]

- **XY**
  - 2,1,1,3,1,[0,1]

- **CDEF**
  - 2,1,[2,5]

- **CD**
  - 2,1,[2,3]

- **ZT**
  - 2,1,3,4,1,[0,1]

- **EF**
  - 2,1,[4,5]

Site 4

- **ABCDEF**
  - 2,1,[0,5]

- **ABCD**
  - 2,1,[0,3]

- **ZT**
  - 2,1,3,4,1,[0,1]

- **EF**
  - 2,1,[4,5]

- **XY**
  - 2,1,1,3,1,[0,1]

- **CD**
  - 2,1,[2,3]

- **ZT**
  - 2,1,3,4,1,[0,1]

- **EF**
  - 2,1,[4,5]

- **XY**
  - 2,1,1,3,1,[0,1]

- **CD**
  - 2,1,[2,3]

- **ZT**
  - 2,1,3,4,1,[0,1]

- **EF**
  - 2,1,[4,5]

**Insertion Operations**

- Insert **XY** between **B** and **C**
- Insert **ZT** between **D** and **E**

**Sending of Operations**
Conflict-free Replicated Data Types (CRDT) Take away

• Strong eventual consistency
  • Parallel and unsynchronized updates
  • Deterministic merge
• State-based approach (CvRDT):
  • Monotonic semi-lattice
  • Reliable broadcast (epidemic propagation)
• Operation-based approach (CmRDT):
  • Commutative concurrent updates
  • Causally-ordered broadcast
References

References

References


Materials

Some of the presented materials are coming from:

• “Conflict-free Replicated Data Types (CRDTs) for collaborative environments”, Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski. Keynote WETICE 2011
  http://events.telecom-sudparis.eu/wetice/invited_speakers/ShapiroKeyNote.pdf

• “Data Replication and Consistency”, Claudia-Lavinia Ignat. Lectures at Université de Lorraine
  https://members.loria.fr/Clignat/replication/
Thank you

COAST Team

http://team.inria.fr/coast/