Conflict resolution in eventual consistency

COS 418: Distributed Systems
Lecture 9
Michael Freedman

[Selected content adapted from M. Shapiro and I. Stoica]

Eventual consistency

- **Eventual consistency**: If no new updates to the object, *eventually* all accesses will return the last updated value

- Common: git, iPhone sync, Dropbox, Amazon Dynamo

- Why do people like eventual consistency?
  - Fast read/write of local copy of data
  - Disconnected operation

Concurrent writes can conflict

- Encountered in many different settings:
  - Peer-to-peer (Bayou)
  - Multi-master clusters (Dynamo)

- Potential solutions
  - “Last writer wins”
    - Thomas Write Rule for DBs with timestamp-based concurrency control: Ignore outdated writes
  - Application-specific merge/update: Bayou, Dynamo

Towards generality?
Consider banking (double-entry bookkeeping):
- Initial: Alice = $50, Bob = $20
- Alice pays Bob $10
  - Option 1: set Alice to $40, set Bob to $30
  - Option 2: decrement Alice -$10, incremental Bob +$10
    - #2 better, but can’t always ensure Alice >= $0

• Works because common mathematical ops are
  - Commutative: \( A \circ B = B \circ A \)
  - Invertible: \( A \circ A^{-1} = 1 \)

General approach: Encode ops as incremental update

Consider shared word processing

• How do I insert a new word?
  - Send entire doc to server? Not efficient
  - Send update operation!

  \[
  \text{insert (string, position) = insert(“1500s”, 166)}
  \]

  Warning: Insert (rather than replace) shifted position of all following text

Operations must be commutative

\[
\begin{align*}
\text{Withdraw} &: \text{$10} & \text{Deposit} &: \text{$15} \\
\text{Deposit} &: \text{$15} & \text{Withdraw} &: \text{$10} \\
\text{Insert} &: (“1500s”, 166) & \text{Delete} &: (1, 0)
\end{align*}
\]
Operations must be commutative

Withdraw $10
$30
Deposit $15
$40
Withdraw $10
$45
Deposit $15
$30
Insert (“1500s”, 166)
A
Delete (1, 0)
B
Delete (1, 0)
C
Delete (1, 0)
D
Insert (“1500s”, 166)

PROBLEM!

Operations must be commutative

Withdraw $10
$40
Deposit $15
$30
Withdraw $10
$45
Deposit $15
$55
Delete (1, 0)
A
Delete (1, 0)
B
Insert (“1500s”, 166)
C
Insert (“1500s”, 166)
D
[ delete 1 char as pos 0 ]

Operational Transformation

Pioneered in GROVE (GRoup Outline Viewing Edit)
C. Ellis and S. Gibbs, 1989

Now found in Apache Wave & Google Docs
State of system is $S$, ops $a$ and $b$ performed by concurrently on state $S$

Different servers can apply concurrent ops in different sequential order

- Server 1:
  - Receives $a$, applies $a$ to state $S$: $S \circ a$
  - Receives $b$ (which is dependent on $S$, not $S \circ a$)
  - Transforms $b$ across all ops applied since $S$ (namely $a$): $b' = \text{OT}(b, \{a\})$
  - Applies $b'$ to state: $S \circ a \circ b'$

- Server 2:
  - Receives $b$, applies $b$ to state: $S \circ b$
  - Receives $a$, performs transformation $a' = \text{OT}(a, \{b\})$
  - Applies $a'$ to state: $S \circ b \circ a'$

Servers 1 and 2 have identical final states: $S \circ a \circ b' == S \circ b \circ a'$

Operational Transformation (OT)

(Used in Google Docs, EtherPad, etc.)
Operational Transformation (OT)
(Used in Google Docs, EtherPad, etc.)

Alice
State: ACE
Ops: ins "ABC" ins "DE" del 2 del 3
  T
  del 4

Bob
State: ACE
Ops: ins "ABC" ins "DE" del 4 del 2
  T
  del 2

Server

More rigorous approach:
Conflict-free replicated data type
Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski 2011

Definition of EC vs Strong EC

• **Eventual delivery:** An update delivered at some correct replica is eventually delivered to all correct replicas

• **Termination:** All method executions terminate

• **Convergence:** Correct replicas that have delivered the same updates eventually reach equivalent state
  • Doesn’t preclude roll backs and reconciling

• **Strong Convergence:** Correct replicas that have delivered the same updates have equivalent state

State-based approach

An object is a tuple \((S, s_0, q, u, m)\)

• Local queries, local updates
• Send full state on receive, merge
  • Update is said ‘delivered’ at some replica when it is included in its casual history
• Causal History: \(C = [c_1, ..., c_n]\)
  • where \(c_i\) goes through a sequence of states: \(c_i^0, ..., c_i^k\) ...
State-based replication

- Local at source $s_1, u(a), s_2, u(b), ...$
  - Precondition, compute
  - Update local payload

  Causal History:
  - on query: $c_i^k = c_i^{k-1}$
  - on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$

Convergence
- Episodically: send $s_j$ payload
- On delivery: merge payloads

Causal History:
- on merge: $c_D^k = c_D^{k-1} \cup c_{DK}^{k-1}$

State-based replication

- Local at source $s_1, u(a), s_2, u(b), ...$
  - Precondition, compute
  - Update local payload

  Causal History:
  - on query: $c_i^k = c_i^{k-1}$
  - on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$

Convergence
- Episodically: send $s_j$ payload
- On delivery: merge payloads

Causal History:
- on merge: $c_D^k = c_D^{k-1} \cup c_{DK}^{k-1}$
State-based replication

- Local at source $s_1.u(a), s_2.u(b), \ldots$
  - Precondition, compute
  - Update local payload
  - Convergence
    - Episodically: send $s_i$ payload
    - On delivery: merge payloads

Causal History:
- on query: $c_i^k = c_i^{k-1}$
- on update: $c_i^k = c_i^{k-1} \cup \{u_i^k(a)\}$
- on merge: $c_i^k = c_i^{k-1} \cup c_i^{k'}$

Desired property:
- After receiving all updates (irrespective of order), each replica will have same state

Example: Union Set

- $u$: add new element to local replica
- $q$: return entire set
- merge: union between remote set and local replica

Example

- Partial order $\subseteq$ on sets

- $\cup$: U (set union)

- Then, we have:
  - commutative: $A \cup B = B \cup A$
  - idempotent: $A \cup A = A$
  - associative: $(A \cup B) \cup C = A \cup (B \cup C)$
Example

- Partial order ≤ on set of integers
- ⊔ \( : \max( ) \)
- Then, we have:
  - commutative: \( \max(x, y) = \max(y, x) \)
  - idempotent: \( \max(x, x) = x \)
  - associative: \( \max(\max(x, y), z) = \max(x, \max(y, z)) \)

Example: Grow-Only Counter

```java
payload integer[n] P
initial [0,0,...,0] [0,0,...,0]
update increment()
  let g = myId()
P[g] := P[g] + 1
query value() : integer v
  let v = \( \sum_i P[i] \)
merge (X, Y) : payload Z
  let \( \forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i]) \)
```

Example: Positive-Negative Counter

```java
payload integer[n] P, integer[n] N
initial [0,0,...,0], [0,0,...,0]
update increment()
  let g = myId()
P[g] := P[g] + 1
update decrement()
  let g = myId()
N[g] := N[g] + 1
query value() : integer v
  let v = \( \sum_i P[i] - \sum_j N[j] \)
compare (X, Y) : boolean b
  let b = (\( \forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \land \forall i \in [0, n-1] : X.N[i] \leq Y.N[i] \))
merge (X, Y) : payload Z
  let \( \forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i]) \)
  let \( \forall i \in [0, n-1] : Z.N[i] = \max(X.N[i], Y.N[i]) \)
```

Semi-lattice

- Partial order ≤ set \( S \) with a least upper bound (LUB), denoted ⊔
  - \( m = x \cup y \) is a LUB of \( \{ x, y \} \) under ≤ iff
    - \( \forall m', x \leq m' \land y \leq m' \)
    - \( \Rightarrow x \leq m \land y \leq m \land m \leq m' \)
- It follows that ⊔ is:
  - commutative: \( x \cup y = y \cup x \)
  - idempotent: \( x \cup x = x \)
  - associative: \( (x \cup y) \cup z = x \cup (y \cup z) \)
**Monotonic Semi-lattice Object**

- A state-based object with partial order \( \leq \) and the following properties, is 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., \( s \cdot m(s') = s \sqcup s' \)
  3. State is monotonically non-decreasing across updates, i.e., \( s \leq s \cdot u \)

**Convergent Replicated Data Type (CvRDT)**

- Theorem: Assuming eventual delivery and termination, any state-based object that satisfies the monotonic semi-lattice property is **SEC**
- Why?
  - Don’t care about order:
    - Merge is both commutative and associative
  - Don’t care about delivering more than once
    - Merge is idempotent

**Commutative Replicated Data Type (CmRDT)**

- Update-based CRDTs:
  - Sends update operations, not state like CvRDT
- Operations are commutative, but not idempotent
  - System must ensure all ops are delivered to other replicas, without duplication, but in any order
  - Often used in more complex settings for concurrent editing

**Industry Use of CRDTs:**

- **Databases:** Redis, Riak, Facebook Apollo
- **Other:** League of Legends Chat, Soundcloud user stream, TomTom device sync
New Module on Monday:

Replicated State Machines