Scalable consistency for replicated data

Annette Bieniusa
Overview

Replication

Scalable consistency

Limitations and Outlook
Interactive cloud-backed applications

- **Large** amounts of shared mutable data
- **High availability** expectations
- **Low latency** business critical
Server-client abstraction
But in reality ...
Replication!

How to keep the replicas consistent?!
Strongly-consistent replication

1. Update(x,4)
2. Update(x,6)
3. ....

• Requires strong synchronization
• High update latency
• Not scalable
• No partition-tolerance

Annette Bieniusa - Scalable Consistency
Asynchronous update-propagation

- Optimistic
- Low update latency
- High availability
- ... at the expense of weaker consistency
## Consistency Trade-offs

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
<th>Consistency</th>
<th>Performance</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
<td>★★★</td>
<td>★★★</td>
<td>★★★</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes.</td>
<td>★★★</td>
<td>★★</td>
<td>★</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
<td>★★</td>
<td>★★</td>
<td>★★</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
<td>★★</td>
<td>★★</td>
<td>★★</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
<td>★★</td>
<td>★★</td>
<td>★★</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
<td>★★★</td>
<td>★★</td>
<td>★★</td>
</tr>
</tbody>
</table>

Adapted from Doug Terry: Cloud Storage Consistency Explained Through Baseball. Dagstuhl 2013-02
Limitations

• Fisher, Lynch, Patterson (’85)
  Consensus ∩ Deterministic ∩ Asynchronous ∩ Faults = ∅

• Brewer (’00); Gilbert & Lynch (’06)
  Strongly-consistent ∩ Available ∩ Partition-tolerant = ∅
Eventual Consistency

• Availability and performance
• Crash-recovery fault-model
• Update each replica independently
  – Transport to other replicas and apply
• Guaranteed delivery: *eventually*, all replicas receive all updates
  – But: Order of updates may differ!
Example: Replicated Integer

- Decentralized conflict resolution
  - “conflict-free by design”
  - No user interaction required
- How are updates propagated?
- Generic policy vs. data type specific
  - Semantics of concurrent updates based on abstract data type

• Decentralized conflict resolution
  - “conflict-free by design”
  - No user interaction required
• How are updates propagated?
• Generic policy vs. data type specific
  - Semantics of concurrent updates based on abstract data type
**Highest timestamp wins (LWW)**

- Generic policy: Symmetry breaking
- Requires total order on updates
- Transmit data + timestamp (unique, monotonically growing)
- E.g. global clock server, synchronized physical clocks
- Widely used: file systems, key-value stores
Multi-value registers (MVR)

- Generic policy
- No automatic conflict resolution, requires user interaction
- Widely used in version control systems, workflow systems
Op-based counter

- Data-type specific: Integer -> Counter
- ADT with interface: incrBy, decrBy
- Send operation + params to other replicas
  Re-execute operation
- Depending on communication layer
  - Idempotence
  - Commutativity
  - Meta-data to track causality
Causal Consistency

- User-friendly
- Subsumes session guarantees read-my-writes, monotonic reads, etc.
- Strongest always-available consistency
- Implementation: VC, graph. Scale?
Vector clocks

- $V_{i[j]} = \#\text{events of } j \text{ observed by } i$
- $V_{i[j]} - V_{i'[j]} = \text{events of } j \text{ that } i' \text{ should transmit to } i$
- $V(a) < V(b) = a \text{ happened before } b$
- $V(a) < V(b) \land V(b) < V(a) = a \text{ concurrent with } b$
State-based counter

- Send payload (state, VC) + merge
- Don't go backwards $\implies$ partial order
- Apply each update once $\implies$ idempotent
- Merge in any order $\implies$ commutative
- Merge contains several updates $\implies$ associative

Annette Bieniusa - Scalable Consistency
Specification: State-based Counter

**payload** integer[n] P
  initial [0, 0, ..., 0]

**update** increment()
  let g = myID()
  P[g] := P[g] + 1

**query** value() : integer v
  let v = \( \sum P[i] \)

**compare** (X,Y) : boolean b
  let b = ( \( \forall i \in [0, n-1] : X.P[i] \leq Y.P[i] \) )

**merge** (X,Y) : payload Z
  let \( \forall i \in [0, n-1] : Z.P[i] = \max (X.P[i], Y.P[i]) \)
**Specification: State-based CRDT**

record (’pl’, ’ua’, ’qa’, ’r’) stateBasedType =

- \( t \text{ compare} \) :: ”’pl ⇒ ’pl ⇒ bool ”
- \( t \text{ merge} \) :: ”’pl ⇒ ’pl ⇒ ’pl ”
- \( t \text{ initial} \) :: ”’pl ”
- \( t \text{ update} \) :: ”’ua ⇒ replicaId ⇒ ’pl ⇒ ’pl ”
- \( t \text{ query} \) :: ”’qa ⇒ ’pl ⇒ ’r ”

- Compare function defines partial order
- Updates monotonically increase the payload
- Merge function computes least upper bound
State-based Sets

- `add(c)`
- `merge({c})`
- `add(b)`
- `merge({a, b})`
- `add(a)`
- `merge({a, b, c})`

Occasionally send local state to other replicas

Merge received state by type-specific function, e.g. set union
Designing for conflict-freedom

\begin{itemize}
  \item \{a\}.add(a) = \{a, a\} \quad \text{– adds unique instance of } a \text{ (unique color)}
  \item \{a, b\}.remove(a) = \{a, b\} \quad \text{– set hide marker on known instances of } a
  \item \{a, a\}.contains(a) \mid true \quad \text{– true if any visible instance of } a \text{ is present}
  \item \{a, b\}.merge(\{b, c\}) = \{a, b, c\} \quad \text{– set union, preserving hide marker}
  \item \textit{partial order of semi-lattice} \quad \text{– set inclusion on elements and markers}
\end{itemize}
Set semantics

Sequential specification of Set:

\[
\begin{align*}
\{\text{true}\} & \quad \text{add}(e) \quad \{e \in S\} \\
\{\text{true}\} & \quad \text{rmv}(e) \quad \{e \notin S\}
\end{align*}
\]

Commutative \((e \neq f)\):

\[
\begin{align*}
\{\text{true}\} & \quad \text{add}(e) \quad \parallel \quad \text{add}(e) \quad \{e \in S\} \\
\{\text{true}\} & \quad \text{rmv}(e) \quad \parallel \quad \text{rmv}(e) \quad \{e \notin S\} \\
\{\text{true}\} & \quad \text{add}(e) \quad \parallel \quad \text{add}(f) \quad \{e, f \in S\} \\
\{\text{true}\} & \quad \text{rmv}(e) \quad \parallel \quad \text{rmv}(f) \quad \{e, f \notin S\} \\
\{\text{true}\} & \quad \text{add}(e) \quad \parallel \quad \text{rmv}(f) \quad \{e \in S, f \notin S\}
\end{align*}
\]

What about:

\[
\begin{align*}
\{\text{true}\} & \quad \text{add}(e) \quad \parallel \quad \text{rmv}(e) \quad \{????\}
\end{align*}
\]
\{true\} \ add(e) \parallel \ rmv(e) \ \{?\}

1. linearisable?
2. last writer wins?
   \{add(e) < rmv(e) \Rightarrow e \notin S \land rmv(e) < add(e) \Rightarrow e \in S\}\}
3. error state? \ \{\bot_e \in S\}\}
4. add wins? \ \{e \in S\}\}
5. remove wins? \ \{e \notin S\}\}

- Deterministic
- Independent of order of delivery
- Independent of local state
- No synchronisation
Designing for conflict-freedom: add-wins set

- **add(a)**: \{a\} → \{a, a\}
  - adds **unique instance** of a (unique color)
- **remove(a)**: \{a\} → ???
  - set **hide marker** on known instances of a
- **contains(a)**: ??? → ???
  - true if any visible instance of a is present

**Partial order of semi-lattice**

```
{a}.add(a) = {a, a}
{a, b}.remove(a) = {a, b}
{a, a}.contains(a) | true
{a, b}.merge({b, c}) = {a, b, c}
```

- **add(b)**: \{a, b\} → \{a\}
- **remove(b)**: \{b\} → ???
- **merge()**: ??? → ???
- **contains(b)**: ??? → ???

- **merge**({a})
- **merge**(\{a\})
- **merge**(\{b\})

- **merge**(\{a\})
- **merge**(\{a\})
- **merge**(\{b\})
**Specification: State-based OR-Set**

**payload** set $E$, set $T$
  initial $\emptyset$, $\emptyset$

**query** contains (element $e$): boolean $b$
  let $b = (\exists n : (e, n) \in E \land (e, n) \not\in T)$

**update** add (element $e$)
  let $n =$ unique() in $E := E \cup \{(e, n)\}$

**update** remove (element $e$)
  let $R = \{(e, n) : (e, n) \in E \land e = e\}$ in $T := T \cup R$

**compare** $(A, B)$: boolean $b$
  let $b = (A.E \subseteq B.E) \land (A.T \subseteq B.T)$

**merge** $(B)$
  $E := E \cup B.E$
  $T := T \cup B.T$

Annette Bieniusa - Scalable Consistency 27
Correctness of CRDTs

1. Convergence
   If two replicas have seen the same set of updates, do they have the same state?

2. Semantics
   If a replica has seen a given set of updates, what is the state of the replica?
Challenges: Programming model

Specification (add-wins-set):
“Remove operation deletes only elements from the set that have been observed at the replica issuing the remove operation. When concurrently adding the element (again), it will remain in the set.”
Challenge: Programming model

is x in the set?
1. Approach: Concurrent Specifications

Problem:
Not every execution can be expressed as a combination of parallel and sequential operations.

\[ a; (c||d) \]

??

• Problem: Not every execution can be expressed as a combination of parallel and sequential operations.
2. Approach: Merge Specification

\[
\begin{align*}
\{x \notin S_1 \land x \notin S_2\} & \quad S = \text{merge}(S_1, S_2) \quad \{x \notin S\} \\
\{x \in S_1 \land x \in S_2\} & \quad S = \text{merge}(S_1, S_2) \quad \{?\}
\end{align*}
\]

⇒ Not possible to specify merge completely, when pre-conditions depend only on visible state of $S_1$ and $S_2$
# 2. Approach: Merge Specification

\[
\begin{align*}
\{x_i \in S_1 \land x_i \in S_2 \land x_i \in S_{1 \cap 2}\} & \quad S = \text{merge}(S_1, S_2) \quad \{x_i \in S\} \\
\{(x_i \notin S_1 \lor x_i \notin S_2) \land x_i \in S_{1 \cap 2}\} & \quad S = \text{merge}(S_1, S_2) \quad \{x_i \notin S\} \\
\{(x_i \in S_1 \lor x_i \in S_2) \land x_i \notin S_{1 \cap 2}\} & \quad S = \text{merge}(S_1, S_2) \quad \{x_i \in S\} \\
\{x_i \notin S_1 \land x_i \notin S_2 \land x_i \notin S_{1 \cap 2}\} & \quad S = \text{merge}(S_1, S_2) \quad \{x_i \notin S\}
\end{align*}
\]

Add more information to precondition:
- use internal, unique identifier \(i\) for elements.
- use common pre-state \(S_{1 \cap 2}\)

Disadvantages:
- Closer to implementation
- State \(S_{1 \cap 2}\) might not exist in execution
3. Specification based on history

\[ x \in S \iff \left( \exists e_a \in E. \, op(e_a) = add(x) \land \left( \forall e_r \in E. \, e_a \prec e_r \land op(e_r) = remove(x) \right) \right) \]

- **E**: set of all visible events
- **op(e)**: operation executed at event e
- **a \prec b**: Event a happened before event b

- Merges are only indirectly visible (via \( \prec \))
- A specification in this form can be turned into an implementation by recording the events and happens-before relation
Verification of State-based CRDTs in Isabelle/HOL (Zeller et al. FORTE ’14)

• Idea: Provide invariant relating the payload of a replica with the visible update history
• Show that invariant implies specification for all queries, holds for initial payload and under no events, and is preserved by updates and merges
• Verified CRDTs:
  – Counter (Increment-only Counter, PN-Counter )
  – Multi-Value Register
  – Sets (Grow-only-Set, 2-phase Set, Observed-Remove Set (Simple, optimized))
• Found minor bug in an MV-register implementation (Assigning the empty set was not an increasing update)
Related work: Understanding Eventual Consistency
(Burckhardt, Gotsman, Yang [TR ’13, POPL’14])

• General framework for formal and declarative specification of semantics using axioms
• Operation context $C = (op, V, ar, vis)$
• Data type specification $F_{\tau}(C)$ specifies return value
• Correctness proofs via replication-aware simulations
Related work: Verifying Eventual Consistency of Optimistic Replication Systems
(Bouajjani, Enea, Hamza [POPL ’14])

• Based on traces, i.e. poset of operations, where operations submitted to one site are totally ordered
• Specification associates return values of operations with posets of operations (→ local interpretation)
• Covers additionally speculative executions and rollbacks
• Liveness: Correct local interpretations convergence eventually (→ global interpretation)
• Verification by reachability and model checking
Operation-based updates

- All replicas have equivalent state in the end
- Sufficient condition
  - Reliable causal delivery $\Rightarrow$ Vector clocks
  - Concurrent operations commute

Annette Bieniusa - Scalable Consistency
CHALLENGES AND LIMITATIONS WHEN PROGRAMMING WITH CRDTS
Challenges: Causality tracking

• Tracking causality requires in general meta-data $O(#\text{clients})$

• High churn is typical: Clients might crash and never return

• Garbage collecting obsolete information only after consensus

• Trick (see work by Baquero et al.): Employ information from transport layer
Challenges: Atomic updates

• Scenario: Friendship management
  – Requests shouldn’t get lost

• Scenario: Virtual wallet
  – User can exchange (virtual) currency for vouchers, game items, ...
  – No money lost!
  – No voucher used twice!

• Operation should be atomic across keys (-> transactions)
Challenges: Bounded divergence

• Scenario: Ad counting
  – Advertisement should be displayed a limited number of times to users in a certain area / country

• Keeping track of how often it is displayed requires counters to deal with high contention

• Estimated count of delivered ads should not diverge too much from actual number

• For a wallet: No divergence!
Summary

- **Replicating** data requires **consistency** handling
- High **availability** precludes strong synchronization
- Tame eventual consistency by deterministic, sound “conflict” handling
- **Replicated Data Types** with state-based / op-based update propagation
  - Counters, Registers, Sets, Maps, Graphs
Acknowledgements

Marek Zawirski  Inria & UPMC-LIP6
Nuno Preguiça   U. Nova de Lisboa
Sérgio Duarte   U. Nova de Lisboa
Valter Balegas  U. Nova de Lisboa
Carlos Baquero  U. do Minho & HASLab
Marc Shapiro    Inria & LIP6
Peter Zeller    TU Kaiserslautern
Chris Meiklejohn Basho

And probably a few more...