Write Fast, Read in the Past: Scalable Consistency all the way to the client

Marc Shapiro

Joint work with:
Marek Zawirski, Inria & LIP6
Annette Bieniusa, U. Kaiserslautern
Valter Balegas, U. Nova Lisboa
Sérgio Duarte, U. Nova Lisboa
Carlos Baquero, U. Minho
Nuno Preguiça, U. Nova Lisboa
Acknowledgments
Geo-replicated cloud storage

[Diagram showing a process with the following steps:
1. Request
2. Process request & store update
3. Reply
4. Transmit update]

1. Request
2. Process request & store update
3. Reply
4. Transmit update to ad-hoc app interface

[Write Fast, Read in the Past: Scalable Consistency all the way to the client]
SwiftCloud approach

App + database at/near client
Update shared store locally
Availability + consistency
Requirements & challenges

Large-scale replication
• Partial replication of data
• … and of metadata
• Small, bounded metadata

High availability □ causal consistency
• Eventual exactly-once delivery
• Causal order versions
• … despite partial replication
• … despite transient failures
• … despite permanent failures

Convergence
Causal gap

Alice @home

Alice @phone

Bob

Bob no photos

post photo

Gap
Causal Consistency

$\nu$ observed effects of $u$ at source

$\square$ $v$ to be delivered after $u$

No gaps

Transitive closure

Strongest always-available consistency
Tracking causality
- Dependence: piggy-backed on update message
- Delivery: compare with incoming message

Representations:
- Graph: worst-case cost, transitive closure cost
- Vector: compact transitive closure, constant cost + optimisations

Issue: size(vector) = $\mathcal{O}$ (#masters)

Log: pruning not contingent on clients
Causal consistency at DC

Full replica state:
- Causally consistent
- Transitivity closed

Vector clock
- size = $O(\#\text{DCs})$
Causal consistency at client

Partial replica: cache interest set
Communicates with DC only
State = DC state | interest set
∪ client updates
Size(vector) = O(#DCs)!
Causal consistency & fail-over

At least once: stubborn
At most once: filter out duplicates
Consistency with new DC
Update identification

Causal order: Vector timestamp VT
No duplicates: client-assigned unique ID CID
Union of vector timestamps
Inconsistency with new DC

Causal gap at client: \( y.\text{add}(C) \) depends on \( x.\text{add}(T) \)

Oregon cannot satisfy: fail-over impossible; client blocks until Ireland recovers

[Write Fast, Read in the Past: Scalable Consistency all the way to the client]
Solution: reading in the past

Client reads only updates that are K-stable □ no gaps
• $K=1$: always fresh, fail-over problematic
• $K=N$: fail-over guaranteed, staleness
  ‣ updates possibly buffered indefinitely

Default: $K=2$, covers common failures

Also improves response time
Converging concurrent updates

[Performance vs. programmability in large-scale distributed systems]
Convergence by construction

Merging concurrent updates:
• Deterministic
• Dependent only on delivery poset
• Not on delivery order, local info

Sufficient conditions:
• Delivered updates commute
• (or) Monotonic semi-lattice

CRDTs
## Conflict-free Replicated Data Types (CRDTs)

Encapsulate replication & resolution semantics
Converge by construction

<table>
<thead>
<tr>
<th>Register</th>
<th>Counter</th>
<th>Graph</th>
<th>Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Last-Writer Wins</td>
<td>• Unlimited</td>
<td>• Directed</td>
<td>• Grow-Only</td>
</tr>
<tr>
<td>• Multi-Value</td>
<td>• Restricted non-negative</td>
<td>• Monotonic DAG</td>
<td>• 2P</td>
</tr>
<tr>
<td><em>Set</em></td>
<td></td>
<td>• Edit graph</td>
<td>• Observed-Remove</td>
</tr>
<tr>
<td>• Grow-Only</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 2P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Observed-Remove</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>Map</em></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Write Fast, Read in the Past: Scalable Consistency all the way to the client]
Experiments

Social networking application
- 90% cache hits
Evaluate
- Availability, responsiveness
- Size of metadata
- Latency, throughput

3 DCs in Amazon EC2
1 DC = 1 EC2 medium instance
100 client nodes in PlanetLab
Multiple sessions per node
Cache size: 512 objects
Update caching + Read-In-Past minimize latency

- SOA = application logic & data in DC
- SwiftCloud: application, updates at client, asynchronous commit
- SwiftCloud RIP: read K-stable
- stable updates: read one, write K
Latency vs. throughput

Latency [ms]

Throughput [TPM x 1000]
Staleness for fault tolerance

[Graph showing staleness and latency over time for different social network DB sizes]
Metadata overhead

![Graph showing metadata overhead for SwiftCloud and PRACTI/Depot]
Summary

Fast response, high availability
  • Shared store replicated at client
  • Full consistency guarantees

Availability: Causal consistency is strongest possible
  (+ transactions + red-blue)

Challenges & solutions:
  • Large database: partial replicate data, metadata
  • Small, bounded metadata: DC-based, pruning
  • Fail-over
    ‣ At-most-once: Client ID + merge clocks
    ‣ Avoid gaps: read in the past (K-stable)
  • Convergence: CRDTs