SwiftCloud: Fault-tolerant geo-replication all the way to the edge

Marek Zawirski
INRIA & UPMC-LIP6

Valter Balegas
U. Nova de Lisboa

Annette Bieniusa
U. Kaiserslautern

Nuno Preguiça
U. Nova de Lisboa

Sérgio Duarte
U. Nova de Lisboa

Marc Shapiro
INRIA & UPMC-LIP6

Carlos Baquero
U. do Minho & HASLab
Interactive applications in the cloud

Data shared between users and devices across the globe
Availability + low latency
Under the hood: geo-replication across data centers

1. request
2. process request & store updates
3. reply
4. replicate updates
5. replicate updates

Replication => read throughput + fault tolerance
Geo-replication => lower latency for reads + disaster tolerance
Optimistic " => lower latency for updates + high availability
... at the expense of weaker consistency

Shapiro – SwiftCloud: FT geo-replication to the edge
Client-DC latency ≈ 10–100ms

1. request: publish a post
2. process
3. reply: ack

Geo-replication guarantees
Client-DC failure anomalies

1. request: get new friends’ posts
2. process
3. reply: post
4. request: like / share the post
5. process
6. error: unknown post!

Geo-replication guarantees

Is my action lost?!?

Corner case logic

Execute-once?
Session guarantees?

Shapiro – SwiftCloud: FT geo-replication to the edge
Client-DC disconnection not supported

Geo-replication guarantees

Inconsistency-prone app-level logic?
Growing demand!
Move application logic, data close to the user

Locality reduces latency, enables offline mode

User-side replicas help with fault tolerance, consistency

- Strongest asynchronous consistency model?
- Design asynchronous commit/propagate protocols
  - Tolerate DC & network faults, disconnect
  - Efficient: small meta-data; scalable with client churn & db size

Out of focus: DC design, security

Shapiro – SwiftCloud: FT geo-replication to the edge
What is the strongest reasonable asynchronous consistency model?
System model

- 10–100 data centres
- \(\gg 10^6\) client-side replicas
- Fast updates, high rates
- WAN latency
- Network failure
- \(\Rightarrow\) asynchronous!
Causal consistency

Causality partial order: client invocation order + reads-from order + transitive closure
Conflict-free Replicated Data Types (CRDTs)

Concurrent updates: deterministic outcome

<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>
<tr>
<td>Set</td>
<td>Graph</td>
</tr>
<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>
<tr>
<td>Map</td>
<td>Sequence</td>
</tr>
</tbody>
</table>
Grow-only set (G-Set)

- **Sufficient condition:** "Monotonic semi-lattice:"
  - Partial order
  - Monotonic
  - *Merge* computes Least Upper Bound
  - *Merge* eventually delivered
Mergeable transaction

Transaction: group of queries and updates

• Writes: all-or-nothing
  • Atomicity extends causality across objects

• Reads: consistent snapshot ($\approx$PSI)
  • Causally closed

• Concurrency control: CRDT $\Rightarrow$mergeable
  – No aborts: asynchronous commit
Designing asynchronous replication protocols

Tolerating DC & network faults, disconnected operation
Asynchronous commit & propagate: at-least-once

Shapiro – SwiftCloud: FT geo-replication to the edge
Partial replication = trouble

Shapiro – SwiftCloud: FT geo-replication to the edge
Client failover and missing dependencies

Missing dependencies after client’s failover – what can protocol do?
1. Ignore and violate consistency [COPS\&Walter SOSP’11, Eiger…]
2. Avoid: use slow synchronous replication [Spanner OSDI’12]
3. Await dependencies and affect availability [Terry PDIS’94]
4. **Leverage client replica to recover missing dependencies**

![Diagram showing dependencies and failover scenarios.](Image)
Leveraging client replica to recover dependencies

**Problem:** how to synchronize with under causal consistency?

Stable updates are durable at DCs

Dependency origin: y.read \{A, B\}

**Idea:** can we establish dependencies more carefully?
Leveraging client replica to recover dependencies

**Problem:** how to synchronize with under causal consistency?

Dependency origin: y.read \{A, B\}

**Idea:** can we establish dependencies more carefully?

**Yes:** read older version
Failover: read in the past but consistent

Restrict reads to: stable updates $\cup$ client’s own updates
Implemented with multiversioning, DC-to-DC stability protocol

Property 1: unstable update does not depend on unstable update of other client
Property 2: stable updates are transitively closed

Algorithm is safe (reads transitively closed updates) and live (can recover deps)
Failover is not for free: read $K$-stable updates

An update is $K$-stable if known to be replicated in $K$ DCs

- No isolation, low staleness
- May miss dependencies for failover

- Higher staleness / isolation
- Dependencies (mostly) available

SwiftCloud uses $K=2$: tolerates 1 DC failure

Durability relies on a client until update becomes stable
Designing asynchronous replication protocols

- Tolerating DC & network faults, client disconnections
- Efficient: small meta-data
Summarizing updates (1)

Vector clocks (piggy-back on every message)
Process = sequence of updates \( \Rightarrow \) 1 number / process
Summarizing updates (2)

Vector clocks (piggy-back on every message)
Process = sequence of updates ⇒ 1 number / process

![Diagram showing vector clocks and processes]

Insight: 1 entry per committing DC [Walter SOSP’11, …]
Commits sequential at DC => assign commit timestamp, summary: number/DC

![Diagram showing commits and timestamps]
Ensuring “at-most-once”

Solution: separate transaction ID from dependency ID
- Transaction ID: assigned by client
- Commit vector: assigned by DC
- Multiple commit vectors OK: merged

1 update, 2 commit vectors, received twice
- Problem for non-idempotent updates
Applications & evaluation

SwiftCloud: 20 kloc + libs
SwiftStore
  • TPC/W
SwiftDocs
  • CRDT map + Sequence
SwiftSocial
  • CRDT Set and LWW-Register, 650 LOC
  • Login / logout
  • Post

• Update profile
• Send/accept friendship request
• Read user’s profile / wall
• Read list of user’s friends

Evaluation: SwiftSocial
  – 25K users, 25 friends each
  – 90% reads, 10% updates
  – Locality: 90% transactions on friend's / own data
Experimental setup

3 DCs in Amazon EC2
- 1 DC = 1 EC2 medium instance
- Persistence: BerkeleyDB

100 client nodes in PlanetLab
- Multiple sessions per node
- Cache: LRU w/DC invalidations
- Cache size: 512 objects

[Map diagram showing distances between Oregon, N. Virginia, Ireland, and PlanetLab nodes]
Update caching + client-assisted FT minimize latency

- SOA = application logic & data in DC
- SwiftCloud: application, updates at client, asynchronous commit
- SwiftCloud FT: read K-stable
- Naïve FT: read one, write K

Operations with > 1 cache miss & selective queries (⇒ move to DC)
Caching gains depend on data locality

CRDTs, mergeable transactions: writes are always local
Cost of fault tolerance

Reading $K$-stable versions has little impact on staleness
Does fault tolerance work?

![Diagram showing latency over time with DC1 highlighted.](image-url)
Does it scale with #data centers? Does it need to?

With caching, increasing #DCs does not improve throughput/latency in saturation

**Reason:** reads answered in cache, bottleneck in updates processing & dissemination

**New opportunities:** horizontal scalability within DC more important than #DCs?

**Win:** edge => more cost-efficient infrastructure (condition: consumed invalidations?)
Conclusion

SwiftCloud brings data, application to the client

- Leverages CRDTs $\Rightarrow$ concurrent updates mergeable
- Asynchronous, e.g. no consensus
- Causal + transactional guarantees
- Better response + disconnected operations (assuming sufficient locality)
- Better performance $\Rightarrow$ lower infrastructure costs

Causal & transactional guarantees

- Safety, liveness: $K$-stable reads, client-assisted failover
- Reliable, fast: separate causality tracking from version identification
SyncFree project  
2013—2016

5 academic + 3 industrial partners

CRDT-based mobile computing in the real world

– Programming models & proofs
– Partial replication
– Security

Hybrid model:
• DCs + ISP points-of-presence + user-side
• Adaptation of the causal consistency algorithm to client + web service model