I Can’t Believe It’s Not Causal!
Scalable Causal Consistency with No Slowdown Cascades
Syed Akbar Mehdi, Cody Littley and Natacha Crooks, UT Austin; Lorenzo Alvisi, UT Austin and Cornell; Nathan Bronson, Facebook; Wyatt Lloyd, USC
14th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’17)

**PROBLEM**

Key Adoption Hurdle: Slowdown Cascades

- Implicit Assumption of Current Causal Systems
- Reality at Scale
- Slowdown Cascade

<table>
<thead>
<tr>
<th>Causal Consistency</th>
<th>In Theory</th>
<th>In Practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventual Consistency</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Causal Consistency</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Strong Consistency</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- The largest web applications use eventually consistent datastores
- Examples: Espresso, TAO, Manhattan

**SOLUTION**

Observable Causal Consistency

Causal Consistency guarantees that each client observes a monotonically non-decreasing set of updates (including its own) in an order that respects potential causality between operations.

Key Idea: Don’t implement a causally consistent data store
Let clients observe a causally consistent data store

Implementing Observable Causal Consistency

1. Our solution is a system called OCCULT (Observable Causal Consistency Using Lossy Timestamps)
2. Each client maintains metadata (called a Causal Timestamp) to encode the most recent snapshot of the datastore it has observed
3. Writes replicate asynchronously without any buffering
4. On reads clients use the causal timestamp to detect whether a shard is safe to read from

Scalable Distributed Transactions

1. OCCULT is the first causal system to support general purpose read-write transactions!
   a) And still no slowdown cascades!
2. Transactions have the following properties:
   a) (Observational) Atomicity
   b) (Observational) Reads from a causally consistent snapshot
   c) No concurrent conflicting writes
3. Transactions are scalable
   a) No centralized timestamp authorities (or sequencers)!
   Transactions are ordered using causal timestamps
   b) Transaction commit latency is independent of the number of replicas!

**Transactions Example**

- \( a = [\text{Bus}] \)
- \( a = [] \)
- \( w(a) = [\text{Abe}] \)
- \( w(b) = [\text{Bob}] \)
- \( \text{Read Set } b = [\text{Bob}] \)
- \( \text{Commit } T_a \)
- \( \text{Read Set } a = [\text{Bus}, \text{Cal}] \)
- \( w(2) = [\text{Bob}, \text{Cal}] \)
- \( \text{Commit } T_b \)

**Evaluation**

OCCULT implemented by modifying Redis Cluster

- Latency Overhead over Redis Cluster
- Goodput evaluated on a heavily contended, zipfian, read-heavy YCSB workload

**Slowdown Cascade Example**

- Writes causally ordered as \( W_1 \rightarrow W_2 \rightarrow W_3 \)
- Replicated write buffers grow arbitrarily because Eiger enforces consistency inside the datastore

**Causal Timestamp Compression**

1. Use local clock of a partition to increment shardstamps.
   a) Keeps shardstamps loosely synchronized despite varying write rates on shards
2. Use high resolution for recent shardstamps and coalesce the rest

**Slowdown Cascades in Eiger (NSDI ’13)**

- Shardstamps: 400, 3989, 3880, 172, 1678
- Shard IDs: 45, 89, 34, 123, *

- Shardstamps: 400, 3989, 3880, 172, 1678
- Shard IDs: 45, 89, 34, 123, *

- Shardstamps: 400, 3989, 3880, 172, 1678
- Shard IDs: 45, 89, 34, 123, *

- Shardstamps: 400, 3989, 3880, 172, 1678
- Shard IDs: 45, 89, 34, 123, *