Transactional storage for geo-replicated systems

Nurbolat Tazhkenov, Ajesh Vaghela

UCL Computer Science

COMPM038/COMPGZ06
8th march 2013
Today:
• Maintaining data consistency and low latency. Problems and goals.
• **Walter.** Key value store system for geo-replicated apps.
• **Parallel Snapshot Isolation** as a key feature of Walter.
  ✧ but before: what is the Snapshot Isolation.
  ✧ Specification of PSI. key properties.
  ✧ PSI with csets and its usage operations. Anomaly.
• Durability and availability in Walter.
• Design and implementation of Walter.
• Results: Walter vs. Berkeley DB, Walter vs. Redis
• Related works.
• Conclusion and future work.
Problems:

1. Write-write conflicts
2. Weak concurrent transactions
3. Long response time. (High latency!)

Goals:

• To support strong concurrent transaction, balance between the following (atomicity, consistency, isolation, durability, coordination)
• Replicated data across distant sites (geo-replicated storage system: for locality, availability, disaster tolerance)
Problems:

1. Write-write conflicts
2. Weak concurrent transactions
3. Long response time. (High latency!)

Goals:
• To support strong concurrent transactions, balance between (atomicity, consistency, isolation, durability, coordination)
• Replicated data across distant sites (geo-replicated storage system: for locality, availability, disaster tolerance)

Why transactions?
To achieve above-mentioned goals:

Walter: provides transaction. New property: Parallel Snapshot Isolation. PSI: replicates data asynchronously and prevents write-write conflicts;

Two techniques:

- Preferred site: no conflicts and low latency
- Counting sets (csets): No conflicts across different sites.

More later!
Walter: Key features to support geo-replicated applications:

- Asynchronous replication across different sites.
- Update-anywhere for certain objects.
- No conflict-resolution logic needed. *Achieved with PSI.*
- Strong isolation within each site. *PSI property.*
What is the Snapshot Isolation:

SI - isolation condition provided by Oracle, SQLServer and so on.

**Provides:**
- Transactions read from snapshot with single commit ordering of transactions.
- Causal ordering: If T2 reads T1, then T1 is ordered before T2 at every time
- If write-write conflict occurs between concurrent transactions, one is aborted

**Two key properties:**
- P1. All operations read the recent committed version, since transaction began.
- P2. (NO write-write conflicts) The write sets of each pair of committed concurrent transactions must be disjoint.
What is new in Parallel Snapshot Isolation:

- Extension of SI: allows different commit orders.

**Key properties of PSI:**

P1. All operations read the recent committed version at *the transaction’s site.*

P2. (NO write-write conflicts) The write sets of each pair of committed *somewhere-concurrent* transactions must be disjoint.

P3. (Commit Causality Across Sites) If a transaction T1 commits at a site A before a transaction T2 starts at site A, then T1 cannot commit after T2 at any site.

Example:
Site A executes T1, T2
Site B executes T3, T4
PSI allows:
Order at site A: T1, T2, T3, T4
Whereas at site B: T3, T4, T1, T2

Why needed?
Two techniques to avoid write-write conflicts and implement PSI:

- **Preferred sites**: each object has a preferred site, no need to check write conflicts.
- **Conflict – free counting set objects**: Objects are modified frequently from many sites, no reason for preferred sites.
  
  Transactions in Walter: read, write and operation on counting sets csets: during concurrent access to object, **no write – write conflicts**.

*How it works:*
- Keeps counter for each element. Mapping from IDs to counts, may be negative.
- Mapping shows occurrence of element in the sets.
- Operation on csets:
  - Add(x) to add element, increments the counter of x
  - Rem(x) to remove x, which decrements the counter of x.
  - Both add and rem commute, no operations conflict

<table>
<thead>
<tr>
<th>Site A:</th>
<th>add(x), add(y) rem(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Site B:</td>
<td>rem(x), add(x), add(y)</td>
</tr>
</tbody>
</table>

*The final result is the y=1*;
Using PSI:

Anomaly in Isolation Property:

<table>
<thead>
<tr>
<th>Anomaly</th>
<th>Snapshot Isolation</th>
<th>PSI</th>
<th>Eventual consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dirty read</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Non-repeatable read</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Lost update</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Short fork</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Long fork</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Conflicting fork.</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Si and PSI:
Read from snapshot $\Rightarrow$ no dirty and non-repeatable reads
No write-write conflicts $\Rightarrow$ no lost updates
But, They allows:
State to fork: the reason is many transactions may read from the same snapshot, make concurrent updates.

NO conflicting fork!
Anomaly cont’d:

**Short fork:**
Concurrent disjoint updates → state to fork
After commit: the state is merged back:
Initially: A=B=0
T1 reads A=B=0 and writes A← 1 and commits.
T2 reads A=B=0 and writes B← 1 and commits.
*Subsequently, T3 reads A=B=1;*

**Long fork:**
Initially: A=B=0
T1, T3 reads A=B=0 and T1: A← 1, T3: B← 1 Commit
T2 reads A=1, B=0, T4 reads A=0, B=1;
*Subsequently, T5 reads A=B=1;*

**Conflicting fork:**
A=0; at time t: T1 writes A←1 and T2 writes A←2, both commits.
Application-specific logic is needed to make A=3.
Durability and availability:

Normal Durability: at site A: if transaction commits at site A then it is stored at replicated cluster storage system. Explanation: writes are not lost due to power failures.

Disaster-safe Durability: writes logged at f+1 sites. f is the desired fault tolerance level.

Cross-site network issues:  
(if there are some committed but not propagated to other sites transactions)  
Two choices in this case:

Conservative: waits for the failed site comes back online.

Aggressive: Sacrifice the durability of a few committed transactions at failed site for better availability.
Basic Architecture

Configuration service

Site 1 server

Site 2 server

Client 1

Client 2

……

Client n
**Implementation**

Implementation:
- C++
- PHP

Experimental Setup:
- WaltSocial
- ReTwis

- Uses Classes
- Several types of functions.
- Background garbage features.

- Amazon EC2
- Private cluster
- Berkeley DB 11gR2
Results & Evaluation

- Walter vs Berkeley DB?
- Fast Commits on normal objects & Csets
- On normal objects:
  Write throughput is slower than read.
  Reason: lock contention in Walter servers
- On csets objects:
  mostly same as normal objects.
  Reason: csets modification use the same fast commit.

<table>
<thead>
<tr>
<th>Name</th>
<th>Read Tx throughput</th>
<th>Write Tx throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walter</td>
<td>72 Ktps</td>
<td>33.5 Ktps</td>
</tr>
<tr>
<td>Berkeley DB</td>
<td>80 Ktps</td>
<td>32 Ktps</td>
</tr>
</tbody>
</table>

Walter: read throughput is slower than BDB
Reason: extra time for local lock and timestamp
WaltSocial performance:

- Tested with 400,000 users.
- Each user 10 status updates, 10 posting.

*The read-only has the highest throughput.*

**Reason:**

Just a read-only transaction. Other operations have lower throughput since they issue cross site communication.
Results & Evaluation cont’d:

ReTwis performance:
• Walter vs. Redis.
• Sites involved:
  Walter – one or two.
  Redis – just one site (no updates from different sites)
• With one site: performance is same.
• Walter with many sites: Higher throughput due to *scalability* across multiple sites.
Related works:

- **Transactions in data centers:** Bigtable, Sinfonia for a single data center only.
  In contrast to several data centers regarding transactions:
  Dynamo – no transaction
  PNUTS – one-record only transaction
  COPS – read-only transaction

- **Database replication and concurrency control:**
  Eagler replication (updates propagate to the replicas before commit) with distributed two-phase locking: P. A. Bernstein *et al.*

- **Disconnected area transaction:**
  Master copy (primary copy) scheme: two-tier replication algorithm to propose tentative update transaction: J. Gray, et al.
  In contrast, Walter: much less work on manually repair or reconciliation.
Conclusion and Future work

• **Strengths:**
  - Supports transaction (ACID)
  - Achieving disaster safe data replication, which is durable
  - No write-write conflict resolution needed
  - Asynchronous data replication (PSI property)

• **Weaknesses:**
  - Latency increases due to cross site transaction.
  - Handling failure: in conservative way: unavailability of some updates.
    in aggressive way: violates PSI property!

• **Questions and future work:**
  - How can a preferred site changed if location of the user is changed?
  - Hybrids between eager and lazy replication: global graph to control graph. T. Anderson, Y. Breitbart *et al.*