

Replication

Anastassia Ailamaki
<http://www.cs.cmu.edu/~natassa>

Overview

Problem:
update anywhere-anytime-anyway transactional replication has unstable behavior as workload scales up

Motivation:
data is replicated for performance **and** availability

Options for distributed systems with replication?



2

Eager or Lazy Replication?

- Eager replication:
keep all replicas synchronized by updating all replicas in a single transaction
- Lazy replication:
asynchronously propagate replica updates to other nodes after replicating transaction commits
- Pros and cons?



3

Ways to Regulate Updates

- Group:
any node with a copy can update that copy.
- Master:
each object has a master node, only master can update primary copy, all other copies are read-only. Other nodes wanting to update must request this from the master.

4

Basic Analysis

- Number of concurrent transactions originating at a node:
 $\# \text{transactions} = \text{TPS} \times \text{Actions} \times \text{Action_Time}$
- N nodes $\Rightarrow N \times \# \text{transactions}$ originate per second.
- Eager system: updates must be replicated to other N-1 nodes \Rightarrow transaction size grows by a factor of N, node update rate grows by N^2 (each Xact is factor of N bigger, and N times more Xacts)
- Lazy system: each user transaction generates N-1 lazy replica updates \Rightarrow N nodes generate N times more transactions each, again N^2 .
- Non-linear growth! Good? Bad? Why?

5

Eager Replication: Problems

- Uniformity assumption everywhere
- Mobile nodes cannot use eager when disconnected \Rightarrow give stale data
 - Solution: Quorum or cluster
- Probability of deadlocks rises very quickly with transaction size and number of nodes (model predicates grows with third power of number of nodes, fifth power of length of XACT – w/o message delays)

6

Lazy Group

XACT commit => send update Xact to every other node

- Use timestamps to detect and reconcile updates:
 - each object carries timestamp of its most recent update
 - each replica update carries new value + tagged with old timestamp
 - if local replica's timestamp = update's old timestamp are equal, update is safe, advance local replica's timestamp.
 - if local replica does not match update's old timestamp, update is "dangerous" and is sent for reconciliation
- Problem: far too many reconciliations, gives rise to "system delusion" (inconsistent, no way to fix it!)

7

Lazy Master

Object owner propagates after update

- No reconciliation; deadlock instead.
- Looks like single node system with much higher transaction rate (WRT deadlock rate).
- Still non-usable by mobile applications
 - Requires contact with object masters
- Still way too many deadlocks (non-scalable)!

8

Non-Transactional Replication

- Idea: Instead of serializability, go for "convergence property": if no new transactions arrive, and all nodes are reconnected, they will all converge to the same replicated state eventually.
- Example: Lotus Notes. Only two ways to update:
 1. Append a timestamped note.
 2. Timestamped replace a value.
- Works well if convergence is only goal; but loses updates, may not be desirable (e.g., checkbook example.)

9

Two-Tier Replication

Goals for ideal replication scheme:

1. **Availability and scalability:** use replication, but avoid instability.
2. **Mobility:** Allow mobile nodes to read and update the database while disconnected.
3. **Serializability:** provide single-copy serializable transactions.
4. **Convergence:** avoid system delusion

10

Two-Tier Replication (cont.)

Two kinds of **nodes**:

- *Mobile* nodes are disconnected, have a replica of the database, originate tentative XACTs.
- *Base* nodes are always connected. Store a replica of database. Most items are mastered at base nodes.

11

Two-Tier Replication (cont.)

- Versions of replicated **data items** at mobile nodes
 1. *Master*: most recent value received from master
 2. *Tentative*: most recent value due to local updates.
- Two kinds of **transactions**
 1. *Base*: work only on master data, produce new master data
 2. *Tentative*: work on local tentative data. Produce
 - new tentative versions
 - a base transaction to be run later on the base nodes

12

Two-Tier Replication (cont.)

- Base transaction generated by tentative transaction may fail or produce different results
- Tentative transaction fails if base transaction doesn't meet *acceptance criterion*

13

Two-Tier Replication (cont.)

How does this differ from reconciliation mechanism of lazy-group replication?

1. The master database is always converged
2. The originating node need only contact a base node to check if tentative transaction is acceptable

14

Mobile Node Actions

When a mobile node connects to a base node, it:

1. discards its tentative object versions (they will be refreshed soon)
2. sends replica updates for any objects mastered at mobile node to the base node
3. sends all its tentative transactions to the base node to be executed
4. accepts replica updates from the base node
5. accepts notice of success or failure for each tentative transaction.

15

Base Node Actions

When contacted by a mobile node, the base node:

1. sends delayed replica update to the mobile node.
2. accepts delayed update from the mobile node.
3. accepts list of tentative transactions, re-runs them
4. after it commits a base transaction, propagates the lazy replica updates as transactions to all other replica nodes.
5. when all tentative transactions have been reprocessed, mobile node's state is converged with base state.

16

Summary: Properties of 2-tier

1. Mobile nodes may make tentative database updates.
2. Base transactions execute with single-copy serializability.
3. Transaction becomes durable when base transaction completes.
4. Replicas at connected nodes converge to the base system state.
5. If all transactions commute, no reconciliations.

17
