15-721 Database Management Systems

Dangers of replication

Instructor: 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?

Eager vs. Lazy Replication

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

Transaction Reconciliation

- needed in lazy replication
- example: bank account with $1,000
- you and your spouse write two $1,000 checks
- *reconciliation* when bank receives checks

- would be nice to automate reconciliation
- bank: *master copy*

Ways to Regulate Updates

- Group replication scheme
  - any node with a copy can update that copy

- Master replication scheme
  - each object has a master node
  - only master can update primary copy
  - all other copies are read-only
  - others wanting to update must request from master

bank account example: works only for bank!

The problem

- update anywhere-anytime-anyway transactional replication is unstable.

- #checkbooks/account*10, reconciliations*1000
- plus disconnect or delay overhead

- scaleup pitfall: system delusion

simple replication vs. global serializability

Basic Analysis

- # concurrent transactions originating at a node:
  #transactions = TPS x Actions x Action_Time
- N nodes => N × #transactions originate per sec.
- *Eager system*
  - updates must be replicated to other N-1 nodes
  - transaction size grows by a factor of N
  - node update rate grows by N^2
- *Lazy system*
  - each user transaction generates N-1 lazy updates
  - N nodes generate N × #transactions each, again N^2.
- Non-linear growth! (=> unstable scaleup behavior)

Eager Replication

- Always consistent
- Mobile nodes cannot use eager when disconnected => give stale data
  - (simple eager: no disconnected updates)
  - get updates from quorum or cluster
- Can still be killed by deadlocks

Eager Replication: Deadlocks

- #locked_objects_by_other_Xtion = (Xtions*actions)/2 (half-way progress)
- P(Xtion requests locked object)= PRL = (Xtions*actions)/(2*DBsize)
- P(Xtion waits) = PW = 1 - (1 - PRL) * actions
- P(cycle of length 2) = PW^2 / #Xtions
- **P(deadlock) 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)**

Lazy Group

XACT commit => send update Xact to all nodes

- Use timestamps to detect and reconcile updates:
  - each object carries timestamp of its most recent update
  - each replica update carries new value + 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!)

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)!

Non-Transactional Replication

Idea: Instead of serializability, "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.

- Lazy group, 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.)

Two-Tier Replication: goals

For an 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

Two-Tier Replication: Idea

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)

Two-Tier Replication (cont.)

- Versions of replicated **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
   - base transaction to be run later on base nodes

Two-Tier Replication (cont.)

- **Base transaction generated by tentative transaction may fail / produce different results**
- **Tentative transaction fails if base transaction doesn't meet *acceptance criterion*, e.g.**
  - The bank balance must not go negative
  - The price quote can not exceed the tentative quote
  - The seats must be aisle seats

Two-Tier Replication (cont.)

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

A:

1. The master database is always converged

2. The originating node need only contact a base node to check if tentative transaction is acceptable

Mobile Node Actions

When mobile node connects to 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 base node
3. sends all its tentative transactions to base node to be executed
4. accepts replica updates from base node
5. accepts notice of success/failure for each tentative transaction

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 base transaction, propagates lazy replica updates as transactions to all other replica nodes
5. when all tentative transactions have been reprocessed, mobile node's state converges with base state

Summary: Properties of 2-tier

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