

CMSC 724: Distributed Transactions; Dangers of Replication

Amol Deshpande

University of Maryland, College Park

March 8, 2007

Distributed Databases

- Goal: a small-scale (dozens of sites) distributed DBMS
 - Location transparency: users don't know where data is
 - Performance transparency: performance independent of submission site
 - Copy transparency: allows replication for availability
 - Transaction transparency: looks like single-site xacts
 - Fragment "transparency": tables can be fragmented to different sites
 - Schema change transparency: schema updates at a single site affect global schema
 - Local DBMS transparency: arbitrary local DBMSs
- Many research prototypes: R*, SDD-1, Distributed Ingres etc. . .

Distributed vs Federated

- Federated: A loose federation of autonomous sites
- Emphasis on “autonomous”
- Little control over what happens where
- Mariposa (@ Berkeley)
 - Became COHERA (bought by Peoplesoft)

- Location transparency
- Local privacy & control
- Local performance on local queries
- Site autonomy: No central catalogs, no central scheduler, no central deadlock detector
- Catalogs stored as “soft state”
- Tables could be fragmented, replicated

Distributed Transactions

- May make updates at many sites
- How do you commit ?
 - Everyone must commit
 - 2-Phase Commit Protocol
- Deadlocks (assuming locking is used) ?
 - Distributed deadlock detection
 - Solution 1: Pass around “waits-for” graphs
 - Solution 2: Time-out based (perhaps more practical)

Distributed Transactions in R*

- Assumptions
 - update in place, WAL
 - batched force of log records
- Desired Characteristics
 - guaranteed exact atomicity
 - ability to “forget” outcome of commit ASAP
 - minimal log writes & message traffic
 - optimized performance in no-failure case
 - exploitation of completely or partially R/O xacts
 - maximize ability to perform unilateral abort
- In order to minimize logging and comm:
 - rare failures do not deserve extra overhead in normal processing
 - hierarchical commit better than 2P

Normal 2-Phase Commit

Coordinator Log	Messages	Subordinate Log
	PREPARE →	
		prepare*/abort*
	← VOTE YES/NO	
commit*/abort*		
	COMMIT/ABORT →	
		commit*/abort*
	← ACK	
end		

- * → forced on log (for durability/atomicity)
- Always log before sending a message
- Total cost:
 - subords: 2 forced log-writes (prepare/commit), 2 messages (YES/ACK)
 - coord: 1 forced log write (commit), 1 async log write (end), 2 messages/subord (prepare/commit)

Normal 2-Phase Commit

- Dealing with failures
 - Recovery process at each site
 - Use logs for handling failures
 - E.g. a subordinate crashes, and wakes up, and sees a prepare* → ask coordinator what happened to the transaction
 - Need to prove correctness carefully
- Hierarchical 2PC
 - Tree-shaped communication hierarchy
 - Non-root, non-leaf nodes behave as both coordinators and subordinates

Presumed Abort/Commit

- Goal: Reduce the number of log-writes and messages
- Presumed abort (PA)
 - Don't ACK aborts, don't force-write logs for aborts
 - After failure and restart, ask the coordinator what happened
 - If coordinator doesn't find any information, it **assumes abort**
 - Much fewer messages with read-only transactions
- Presumed commit (PC)
 - Don't ACK commits, don't force-write logs for commits
 - Some tricky details
 - **Better than PA** if commit is more common than abort
- Can choose on a per-transaction basis

Dangers of Replication

- Gray, Helland, O'Neil, Shasha; SIGMOD 1996
- Replication critical for performance/availability
- How do you keep the copies up-to-date ?
- Two axes:
 - How are copies kept in-sync: Eager vs Lazy
 - Eager: All copies updated in a single transaction; deadlocks can happen
 - Lazy: Updates propagated lazily; reconciliations may be required
 - Who owns the objects: Group vs Master
 - Group: Everyone owns the object and can update
 - Master: Singler owner; must take permission from it to update

Dangers of Replication

- Scaleup pitfall: System works fine on a small demos, dies when you scale up
- System delusion: Replicas diverge too much and can't be reconciled
- **Analysis** of Eager Group Replication:
 - Model the number of transactions, the running time for each transaction etc. . .
 - Use this to compute the probability of “wait”
 - Use this to compute the probability/rate of “deadlocks”
 - Turns out: Deadlocks grow as $O(n^3)$ where n = number of nodes
 - Lower if you assume the database size also scales

Dangers of Replication

- Eager Master: Deadlocks a bit lower
- Lazy Group: The transactions with deadlocks need to be reconciled now, so $O(n^3)$
- Lazy Master: Deadlocks grow as $O(n^2)$
- Etc. . .

A Two-Tier Replication Scheme

- Make a distinction between “base” (always-connected) and “mobile” nodes
- Use eager replication among base nodes
- Mobile nodes make “tentative” transactions when disconnected
 - “Submit” these to some base node when connected
 - The transaction may be rejected
 - Acceptance Criteria:
 - Submit a criteria (e.g. *balance* > 0) when submitting a transaction
 - Even if the results are different, the transaction may be accepted
 - In other words, use “commutative” transactions