Concurrency Control

Transactions and Distributed Transactions
Announcements

• 1) Exam Thursday
• 2) Exam review session
• 3) Homework 1 back
• 4) dga,vrv out until Thursday
Last time: RAID

- Trade capacity for reliability
- Throughput growing...delay not
Today: Concurrency control

• Local concurrency control
  – Transactions
  – Two-phase locking

• Distributed concurrency control
  – Two-phase commit
Transactions

• Fundamental abstraction grouping work
• begin, commit, rollback/abort
  – checkpoint
  – forced rollbacks on error…
ACID properties

• Atomicity: all or nothing
• Consistency: guarantee basic properties
• Isolation: each transaction runs as if alone
• Durability: cannot be undone

– Define: serializability
  • …even though transactions are concurrent
The debit/credit example

```java
bool xfer(Account src, Account dest, long x) {
    if (src.getBalance() >= x) {
        src.setBalance(src.getBalance() - x);
        dest.setBalance(dest.getBalance() + x);
        return TRUE;
    }
    return FALSE;
}
```
Problems to avoid

• Lost updates
  – Another transaction overwrites your change based on a previous value of some data

• Inconsistent retrievals
  – You read data that can never occur in a consistent state
Conflicting operations

- Reads are compatible with reads
- Reads conflict with writes
- Writes conflict with writes
Centralized Locking

• On an object-by-object basis
  – easy enough with the tools we have
• Locks alone aren’t enough…
• Two-phase locking (2PL)
  – 1: acquire locks (gradually or otherwise)
  – 2: release locks (all at once on commit/abort)
Debit/credit with locking

- lock src; lock dest
Deadlocks

• Prevention
  – All locks atomically?
  – All locks in some order?

• Detection
  – cycle-finding algorithm (either greedy or lazy)
    • The “wait-for” graph: who waits for whom

• Ignore
  – abort long-running transactions?
Distributed transactions

• Data stored at distributed locations
• Failure model:
  – messages are delayed/lost
  – servers crash, but can recover saved persistent storage
• Need an atomic commit protocol
The coordinator

- Begins transaction
  - Assigns unique transaction ID
- Responsible for commit/abort
- Many systems allow the client to be the coordinator for its own transactions
The participants

- The servers with the data used in the distributed transaction
Problems with simple commit

- Coordinator broadcasts "commit!" to participants until all reply
  - "One-phase commit"
  - What happens if one of a participant fails? Can the other participants now undo what they have already committed?
Two-phase commit

• Commit itself is two-phases
  – Voting phase
  – Committing phase

• Operations:
  – canCommit?(T) -> yes/no
  – doCommit(T)
  – doAbort(T)
  – haveCommitted(participant, T)
  – getDecision(T) -> yes/no
The voting phase

- `canCommit?(T)`
- Participants must “prepare to commit” using permanent storage before answering yes
  - Objects are still locked, T can be aborted
  - “Prepared state” can be recovered if crash
- Outcome of T is uncertain until `doCommit` or `doAbort`
  - `doCommit` if unanimous, `doAbort` otherwise
The commit phase

- The fate of the transaction is already determined
  - All participants have voted
  - The coordinator knows the result
  - Uncertain participants (or their replacements) will eventually get the doCommit or doAbort message from the coordinator

- Participants use getDecision if they have not heard from the coordinator for a while
2PC sequence of events

Coordinator

“prepared”

canCommit?

Yes

doCommit

haveCommitted

“committed”

“done”

Participant

“prepared” (persistence)

“uncertain” (objects still locked)

“committed”

participant not allowed to cause an abort after it says it canCommit
2PC example
2PL with 2-Phase Commit

- Each participant uses 2PL for its objects, 2PC for the commit process
Distributed deadlock detection

• Centralized?
• Edge-chasing?
• Ignore?
Topics I did not teach follow…
Optimistic concurrency control

• Run transaction without locking
• Commit atomically (in a mutex)
  – Check to make sure that the values you read have not been written since you read them
  – Abort if the values have changed
Timestamp-ordering

• Timestamp transaction at begin
  – “Defines serial order of transactions”
• Timestamp each object with “latest” read so far
• Tentatively write a shadow copy
• Run without locks, check for conflicts as you work
  – Cannot write an object read by a later transaction
  – Cannot write an object write-committed by a later transaction
  – Cannot read an object written by a later transaction
• Abort immediately if you detect a conflict
  – Reads wait until earlier tentative writes commit or abort
  – Never need to wait for a later transaction
Time-ordering write rule

• Must be > maximum-read-time
• Must be > committed write time
Time-ordering read rule

• Must be > committed write time
• Select last write that is earlier than us
  – If it is committed, read
  – If it is not committed, wait for commit/abort and re-read
Timestamp-ordering writes

(All these assume T3 writes, T3 > max read-time)
Timestamp-ordering writes

(All these assume T3 reads)

T2 reads...

T4 reads...

(T3 aborts)
Distributed time-stamp ordering

• Use Lamport clocks to timestamp T
• Still use 2P-commit
  – doCommit almost always “yes”
  – Conflicts usually cause immediate abort
• Deadlocks cannot occur
  – Can only wait for an earlier Transaction => no cycles in wait-for graph