CSE 486/586 Distributed Systems
Concurrency Control --- 3

Steve Ko
Computer Sciences and Engineering
University at Buffalo

Recap

• Strict execution of transactions?
  – *Delay both their read and write operations* on an object until
    all transactions that previously wrote that object have either
    committed or aborted

• Two phase locking?
  – Growing phase
  – Shrinking phase

• Strict two phase locking?
  – Release locks only at either commit() or abort()

Can We Do Better?

• What we saw was “exclusive” locks.
• Non-exclusive locks: break a lock into a read lock
  and a write lock
• Allows more concurrency
  – Read locks can be shared (no harm to share)
  – Write locks should be exclusive

Non-Exclusive Locks

<table>
<thead>
<tr>
<th>non-exclusive lock compatibility</th>
<th>Lock already</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>WAIT</td>
</tr>
<tr>
<td>write</td>
<td>WAIT</td>
<td>WAIT</td>
</tr>
</tbody>
</table>

• A read lock is promoted to a write lock when the
  transaction needs write access to the same object.
• A read lock shared with other transactions’ read
  lock(s) cannot be promoted. Transaction waits for
  other read locks to be released.
• Cannot demote a write lock to read lock during
  transaction – violates the 2P principle

Example: Non-Exclusive Locks

Transaction T1

OpenTransaction()

balance = b.getBalance()

R-Lock B

Commit

Promote lock on B

Transaction T2

OpenTransaction()

balance = b.getBalance()

R-Lock B

Promote lock on B

Cannot Promote lock on B, Wait

2PL: a Problem

• What happens in the example below?

Transaction T1

OpenTransaction()

balance = b.getBalance()

R-Lock B

Promote lock on B

Cannot Promote lock on B, Wait

Transaction T2

OpenTransaction()

balance = b.getBalance()

R-Lock B

Promote lock on B

Cannot Promote lock on B, Wait

b.setBalance = balance * 1.1

Cannot Promote lock on B, Wait

...
Deadlock Conditions

- Necessary conditions
  - Non-sharable resources (locked objects)
  - No lock preemption
  - Hold & wait or circular wait

Preventing Deadlocks

- Acquiring all locks at once
- Acquiring locks in a predefined order
- Not always practical:
  - Transactions might not know which locks they will need in the future
- One strategy: timeout
  - If we design each transaction to be short and fast, then we can abort() after some period of time.

Extracting Even More Concurrency

- Allow writing tentative versions of objects
  - Letting other transactions read from the previously committed version
- Allow read and write locks to be set together by different transactions
  - Unlike non-exclusive locks
- Read operations wait only if another transaction is committing the same object
- Disallow commit if other uncompleted transactions have read the objects
  - These transactions must wait until the reading transactions have committed
- This allows for more concurrency than read-write locks
  - Writing transactions risk waiting or rejection when commit

Two-Version Locking

- Three types of locks: read lock, write lock, commit lock
  - Transaction cannot get a read or write lock if there is a commit lock
- When the transaction coordinator receives a request to commit
  - Converts all that transaction’s write locks into commit locks
  - If any objects have outstanding read locks, transaction must wait until the transactions that set these locks have completed and locks are released
- Compare with read/write locks:
  - Read operations are delayed only while transactions are committed
  - Read operations of one transaction can cause a delay in the committing of other transactions

Two-Version Locking lock compatibility

<table>
<thead>
<tr>
<th>Lock already set</th>
<th>Lock requested</th>
<th>read</th>
<th>write</th>
<th>commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
<tr>
<td>read</td>
<td>OK</td>
<td>OK</td>
<td>WAIT</td>
<td></td>
</tr>
<tr>
<td>write</td>
<td>WAIT</td>
<td>WAIT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commit</td>
<td>WAIT</td>
<td>WAIT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CSE 486/586 Administrivia

- PA3 deadline: 3/29 (Friday)
- PA2 grading
  - Will be done sometime next week (we wanted to grade the midterm first.)
- Anonymous feedback form still available.
- Please come talk to me!
Distributed Transactions
- Transactions that invoke operations at multiple servers

Coordinator and Participants
- Coordinator
  - In charge of begin, commit, and abort
- Participants
  - Server processes that handle local operations

Example of Distributed Transactions

Atomic Commit Problem
- Atomicity principle requires that either all the distributed operations of a transaction complete, or all abort.
- At some stage, client executes closeTransaction(). Now, atomicity requires that either all participants (remember these are on the server side) and the coordinator commit or all abort.
- What problem statement is this?
  - Consensus
  - Failure model
    - Arbitrary message delay & loss
    - Crash-recovery with persistent storage

Atomic Commit
- We need to ensure safety in real-life implementation.
  - Never have some agreeing to commit, and others agreeing to abort.
- First cut: one-phase commit protocol. The coordinator communicates either commit or abort, to all participants until all acknowledge.
- What can go wrong?
  - Doesn’t work when a participant crashes before receiving this message.
  - Does not allow participant to abort the transaction, e.g., under deadlock.

Two-Phase Commit
- First phase
  - Coordinator collects a vote (commit or abort) from each participant (which stores partial results in permanent storage before voting).
- Second phase
  - If all participants want to commit and no one has crashed, coordinator multicasts commit message
  - If any participant has crashed or aborted, coordinator multicasts abort message to all participants
Two-Phase Commit

- Communication

- To deal with server crashes
  - Each participant saves tentative updates into permanent storage, right before replying yes/no in first phase. Retrievable after crash recovery.
  - To deal with canCommit? loss
    - The participant may decide to abort unilaterally after a timeout (coordinator will eventually abort)
  - To deal with Yes/No loss, the coordinator aborts the transaction after a timeout (pessimistic!). It must announce doAbort to those who sent in their votes.
  - To deal with doCommit loss
    - The participant may wait for a timeout, send a getDecision request (retries until reply received) – cannot abort after having voted Yes but before receiving doCommit/doAbort!

Problems with 2PC

- It’s a blocking protocol.
- Other ways are possible, e.g., 3PC.
- Scalability & availability issues

Summary

- Increasing concurrency
  - Non-exclusive locks
  - Two-version locks
  - Hierarchical locks
- Distributed transactions
  - One-phase commit cannot handle failures & abort well
  - Two-phase commit mitigates the problems of one-phase commit
  - Two-phase commit has its own limitation: blocking

Acknowledgements

- These slides contain material developed and copyrighted by Indranil Gupta (UIUC).