CSE 486/586: Distributed Systems

Concurrency Control (part 3)

Ethan Blanton

Department of Computer Science and Engineering
University at Buffalo

Lost Update

- Some transaction T1 runs interleaved with some transaction T2.
- T1 reads the value of some state object *o* and stores it locally as *v*.
- T2 reads and modifies *o*.
- T1 writes *o* based on its stored *v*.

Thus, T2's update of *o* is lost, because T1 never saw it.

Inconsistent Retrieval

- Some transaction T1 runs interleaved with some transaction T2.
- T1 modifies the value of some state object *o*.
- T2 reads state objects *o* and *u* for use in computation.
- T1 modifies the value of some state object *u*.

Thus, T2's retrieval of the shared state is inconsistent.

- *o*: post-T1
- *u*: pre-T1

Serial Equivalence

If an interleaving of two transactions T1 and T2 (without abort) is equivalent to the serial execution of T1 followed by T2, the interleaving has serial equivalence.

If T1 can be aborted, then aborting T1 may require aborting T2. Consider:

1. T1 modifies some state object *o*.
2. T2 reads *o*.
3. T1 modifies some state object *u*.
4. T2 reads *u*.
5. T2 modifies *u*, based on the values of *o* and *u*.

In this example, if T1 and T2 run to completion, it is serially equivalent to T1 followed by T2.

If T1 aborts before line 3, T2 must also abort!

Strict Execution

By delaying visible writes until commit time, the cascading aborts with simple serial equivalence can be avoided.

To achieve this:

- Locks are taken as objects are needed
- No lock can be taken once any lock has been released
- Locks are released as objects are no longer needed

This is Two-Phase Locking.

In strict two-phase locking, locks are released only at commit or abort.

Read-Write Locks

Concurrency can be improved by allowing multiple read locks to be taken simultaneously on an object, but only one write lock.

Read locks can be promoted write locks (by waiting for readers to unlock), but multiple promotions on the same lock can lead to deadlock.

T1	T2
`ReadLock(a)`	`ReadLock(a)`
...	...
`PromoteLock(a)`	...
...	`PromoteLock(a)`

Two-Version Locking

Concurrency can be improved farther by delaying writes to shared state until a transaction commits.

To achieve this, transactions write to a tentative version of an object that is a copy of the shared state.

- Read locks can be taken even when a write lock exists
- Write locks remain exclusive with other write locks
- Upon commit, write locks are promoted to commit locks
- Commit lock promotion waits until all readers unlock

Once commit starts and all write locks have been promoted, the writer copies its tentative version to the shared state.

Distributed Transactions

A distributed transaction invokes operations on multiple servers.

Flat distributed transactions may involve multiple servers, but only one `begin()`/`commit()` pair.

Nested distributed transactions involve both multiple servers and additional transactions with their own `begin()`/`commit()` pairs.

(Of course, they might `abort()` as well!)

Distributed Transaction Roles

Distributed transactions have:

- A coordinator:
  The coordinator is in charge of the `begin()`, `commit()`, and `abort()` operations.
- One or more participants:
  Participants are server processes that handle local operations on state (or perform calculations).

The coordinator may also be a participant.

Commit Atomicity

Commit must be atomic — either all operations must complete, or all operations must abort.

When the transaction is complete:

- The coordinator schedules a commit
- All participants must commit, or
- The commit fails and the coordinator must abort, so
- All participants must abort

When all processes must make a binary decision, what do we have?

Commit Atomicity

Commit must be atomic — either all operations must complete, or all operations must abort.

When the transaction is complete:

- The coordinator schedules a commit
- All participants must commit, or
- The commit fails and the coordinator must abort, so
- All participants must abort

When all processes must make a binary decision, what do we have?

Consensus!

One-Phase Commit

Assume that the system is asynchronous, not byzantine, and that we have a crash-recovery model.

We want to ensure safety with the property that all participants commit, or all participants abort.

In a one-phase commit protocol, the coordinator simply notifies all processes to commit or abort.

Does that work?

One-Phase Commit

Assume that the system is asynchronous, not byzantine, and that we have a crash-recovery model.

We want to ensure safety with the property that all participants commit, or all participants abort.

In a one-phase commit protocol, the coordinator simply notifies all processes to commit or abort.

Does that work?

- A participant cannot abort the transaction (*e.g.*, due to deadlock)
- Participant crashes after the commit decision are not handled

Two-Phase Commit

In a two-phase commit protocol, the commit proceeds in two phases:

- First Phase:
  - The coordinator collects a vote for commit or abort from each participant.
  - Each participant stores the transaction state in permanent storage before voting.
- Second Phase:
  - If any participant has crashed or votes to abort, the coordinator instructs all participants to abort.
  - If no participants have crashed and all participants vote to commit, the coordinator instructs all participants to commit.

Two-Phase Commit Failure Handling

- Participant crashes:
  - Before voting, the transaction aborts
  - After voting, the transaction state can be restored from permanent storage
- Coordinator voting query loss:
  - Individual participants can decide to abort after timeout
  - The coordinator will eventually abort with a timeout
- Vote loss:
  - The coordinator must abort
- Commit message loss:
  - The participant can query the coordinator for a decision until a reply is recieved
  - A participant that has voted no can unilaterally abort
  - A participant that has voted yes must not abort until it hears a commit or abort instruction!

Problems with 2-Phase Commit

- Blocking
- Scalability
- Availability

Summary

- Review of methods to increase concurrency
  - Read-write (non-exclusive locks)
  - Two-version locks
- Distributed transactions
  - One-phase commit
  - Two-phase commit
  - Limitations of two-phase commit (blocking, reduces concurrency)

References I

Required Readings

[1] Textbook. Sections 16.4, 17.1-17.3.

Acknowledgements

These slides are based on slides from Steve Ko, used with permission.

Those slides contain the following attribution:

- These slides contain material developed and copyrighted by
  - Indranil Gupta ant UIUC
  - Mike Freedman and Jen Rexford at Princeton