Locking and Commit Protocols

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo

Locking

So far we have looked at exclusive locks:
One process can lock a lock, all others are blocked.

There are also non-exclusive locks:

- Read/Write locks (many readers, one writer)
- Two-version locks (many readers/writers, writes wait)

In both cases, many readers can proceed simultaneously.

Committing

Distributed commit has its own challenges.

All parties must either commit, or abort.

Committed transactions must be durable.

This despite the fact that any party can fail!

Read/Write Locks

Recall that read-read is not a conflict.

We can increase concurrency by allowing read-read:

- T1 and T2 both wish to read datum *S*
- T1 and T2 both lock *S* for reading
- T1 and T2 proceed concurrently

This restricts concurrency only for write conflicts.

R/W Lock States

Unlike mutexes, Read/Write locks have three states [1]: Unlocked, Read Locked, and Locked

The allowable state transitions are:

State	Read Lock	Write Lock
Unlocked	Read Locked	Write Locked
Read Locked	Read Locked	Block for Unlock
Write Locked	Block for Unlock	Block for Unlock

Read Locked moves to unlocked only when all readers unlock.

Write Locked moves to unlocked only when all writers unlock.

Lock Promotions

Additional concurrency can be allowed with lock promotions.

A process holding a read lock can promote it to a write lock.

This does not unlock the data item!

Lock promotions follow write lock rules.

Demotions from write to read must be prohibited for 2PL.

The drawback of promotions is deadlock.

Lock Promotion Deadlock

T1:
- Read Lock A
- compute
- Write Lock A

T2:
- Read Lock A
- Write Lock A

What if T2 runs while T1 is in "compute"?

Two-Version Locks

Two-Version locks increase concurrency even further.

They allow one writer and many readers to operate concurrently.

They operate like read-write locks, but the first write lock is immediate.

Consistency is maintained by delaying the write.

Two-Version State

A writing transaction writes to a copy with two-version locking.

When the transaction completes, a fourth lock state is used: Committing.

The Committing state is like a Read/Write Lock Write state: it is truly exclusive.

The key is that the write is delayed until all readers finish.

Thus all concurrent reads happen before the write.

Write-Write Conflicts

Write-write conflicts are solved via mutual exclusion.

If two transactions write the same state, one must wait.

If two transactions write different state, but overlap, deadlock may occur:

T1:
- Read Lock A
- compute
- Write Lock B

T2:
- Read Lock A
- Write Lock B

(Again, T2 runs during T1 “compute”.)

Aborting on Deadlock

With both R/W promotion and two-version locking, even strict two-phase locking can lead to deadlock.

There are two solutions:

- Acquire all locks immediately
- Abort on deadlock

If transactions:

- Are expected to be very fast
- Rarely conclit

…then abort-and-retry will normally succeed.

Distributed Transactions

A distributed transaction invokes operations on multiple servers.

They can be flat or nested:

- Flat: may involve multiple servers, but only one begin/commit pair.
- Nested: involve both multiple servers and additional transactions with their own begin/commit pairs.

Aborting a nested transaction cascades.

Distributed Transaction Roles

Distributed transactions have:

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

The coordinator may also be a participant.

Commit Atomicity

Even distributed commit must be atomic!

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 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.

Our safety property is atomic commit/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.

Our safety property is atomic commit/abort.

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

Does that work?

- What if a participant cannot abort the transaction (*e.g.*, due to deadlock).
- What if a participant crashes after the commit decision?

Two-Phase Commit

Two-phase commit [2] fixes this with (surprise) two phases:

First Phase:

- The coordinator collects a votes for commit or abort.
- 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 all participants vote to commit, the coordinator instructs all participants to commit.

Failure Handling I

Failures can occur at four places:

- A participant may crash at any time
- Communication may be lost requesting a vote
- Communication may be lost sending a vote
- Communication may be lost declaring commitment

If a participant crashes:

- Before voting: the transaction aborts
- After voting: the transaction is in permanent storage

Coordinator crashes are somewhat more complicated.

They can be handled!

Failure Handling II

Lost messages:

- Vote requests: the coordinator will time out and abort
- Votes: the coordinator will time out and abort
- Commit confirmation:
  - Participants that voted no can abort.
  - Participants that voted yes must not abort until a resolution is received!

Problems with Two-Phase Commit

FLP dictates indefinite blocking.

The coordinator is a single point of failure.
(Is that fixable? Why or why not?)

Scalability is poor for many parties.

Summary

- Non-exclusive locking can increase concurrency
  - Deadlock and aborts can be triggered!
- Read/Write locks allow multiple readers in parallel
- Two-version locks allow multiple readers and one writer
- Deadlock detection and abort-and-retry can be effective
- Distributed transactions require multi-process atomic commits
- Two-phase commit solves races in a simple commit

References I

Required Readings

[2] C. Mohan and Bruce G. Lindsay. “Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions”. In: *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, Aug. 1983, pp. 76–88. DOI: 10.1145/800221.806711. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.7048&rep=rep1&type=pdf.

Optional Readings

References II

[1] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. "Concurrent Control with "Readers" and "Writers"". In: *Communications of the ACM* 14.10 (Oct. 1971). Ed. by Brian Randell. DOI: 10.1145/362759.362813. URL: https://search.lib.buffalo.edu/permalink/01SUNY_BUF/12pkqkt/cdi_crossref_primary_10_1145_362759_362813.

To retrieve a copy of this material, or related materials, see `https://www.cse.buffalo.edu/~eblanton/`.