

Distributed Transactions with Two-Phase Commit

R&G - Chapter 20



Distributed vs. Parallel?

- Earlier we discussed Parallel DBMSs
 - Shared-memory
 - Shared-disk
 - Shared-nothing
- Distributed is basically shared-nothing parallel
 - Perhaps with a slower network

What's Special About Distributed Computing?

- Parallel computation
- No shared memory/disk
- Unreliable Networks
 - Delay, reordering, loss of packets
- Unsynchronized clocks
 - Impossible to have perfect synchrony
- Partial failure: can't know what's up, what's down

“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable”.
— Leslie Lamport, Turing 2013



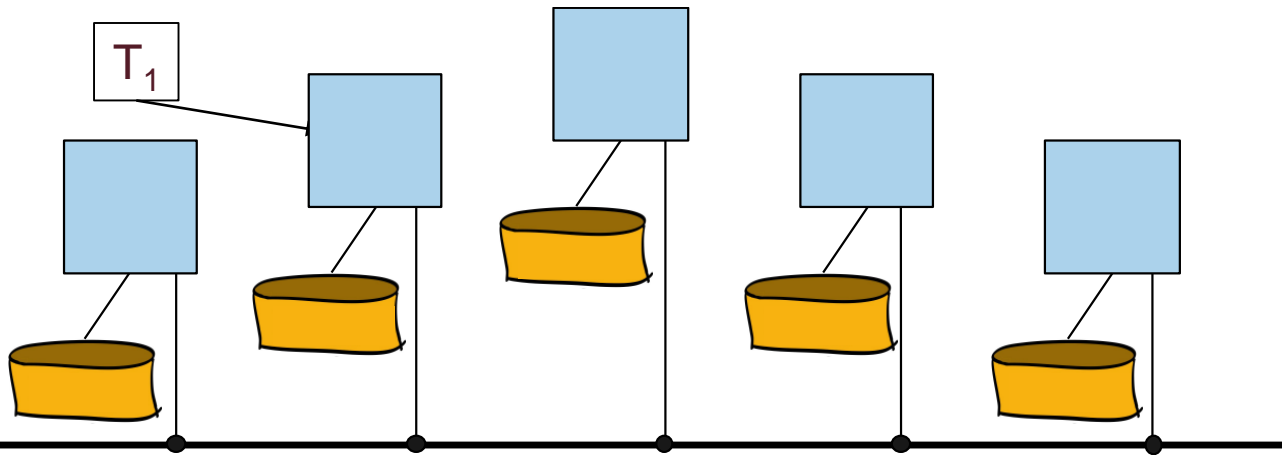
Distributed Database Systems

- DBMS an influential special case of distributed computing
 - The trickiest part of distributed computing is state, i.e. Data
 - Transactions provide an influential model for concurrency/parallelism
 - DBMSs worried about fault handling early on
- Special-case because not all programs are written transactionally
 - And if not, database techniques may not apply
- Many of today's most complex distributed systems are databases
 - Cloud SQL databases like Spanner, Aurora, Azure SQL
 - NoSQL databases like DynamoDB, Cassandra, MongoDB, Couchbase...
- We'll focus on transactional concurrency control and recovery
 - You already know many lessons of distributed query processing

DISTRIBUTED LOCKING

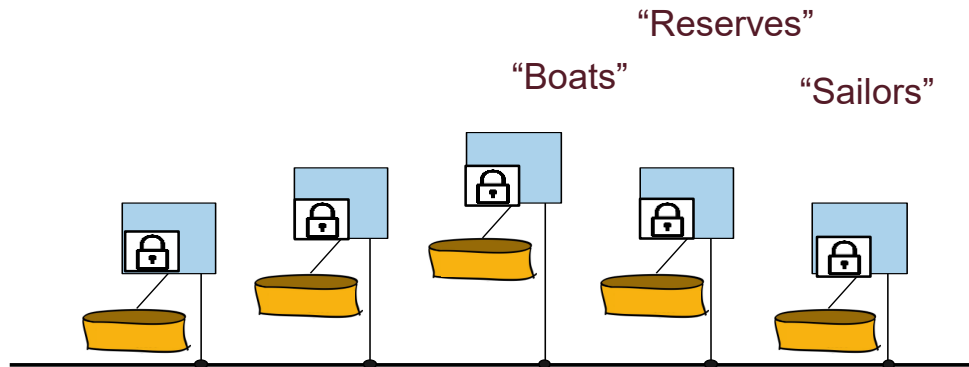
Distributed Concurrency Control

- Consider a shared-nothing distributed DBMS
- For today, assume partitioning but no replication of data
- Each transaction arrives at some node:
 - The “coordinator” for the transaction



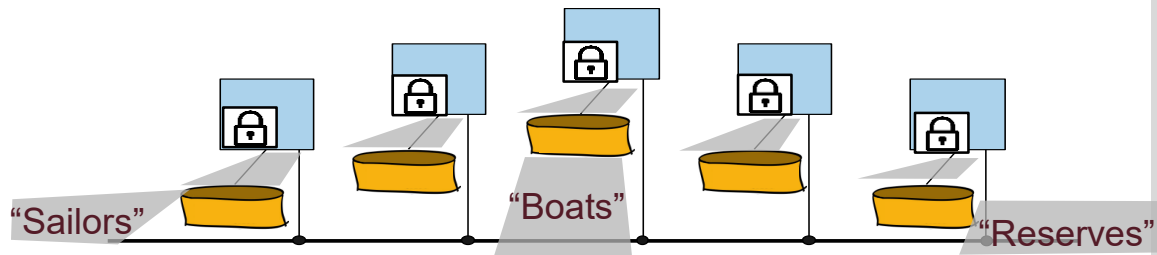
Where is the Lock Table

- Typical design: Locks partitioned with the data
 - Independent: each node manages “its own” lock table
 - Works for objects that fit on one node (pages, tuples)
- For coarser-grained locks, assign a “home” node
 - Object being locked (table, DB) exists across nodes



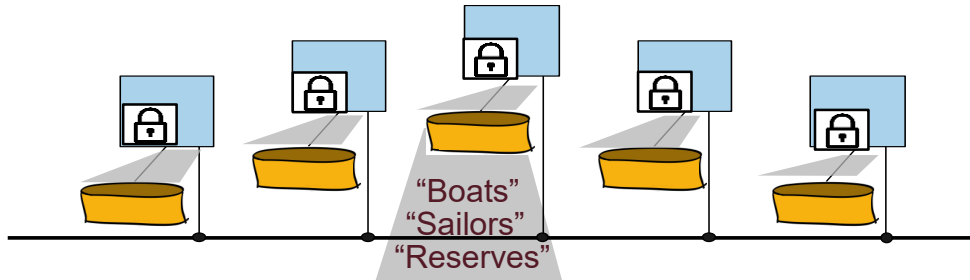
Where is the Lock Table, Pt 2

- Typical design: Locks partitioned with the data
 - Independent: each node manages “its own” lock table
 - Works for objects that fit on one node (pages, tuples)
- For coarser-grained locks, assign a “home” node
 - Object being locked (table, DB) exists across nodes
 - These locks can be partitioned across nodes



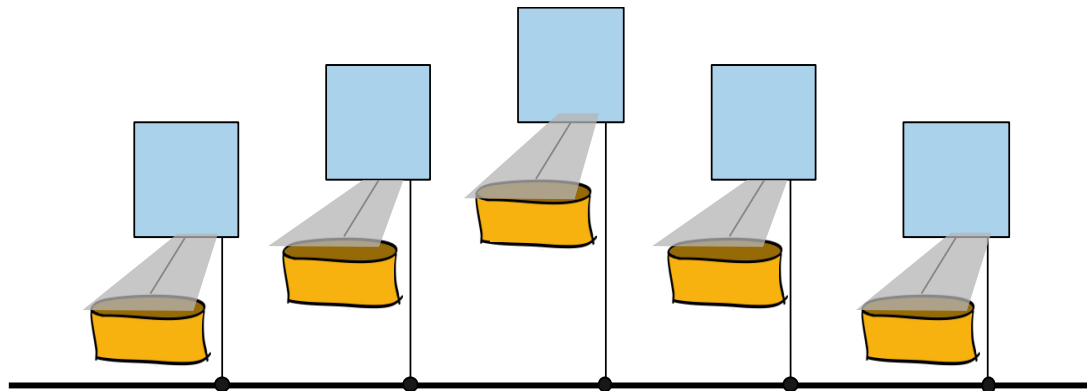
Where is the Lock Table, Pt 3

- Typical design: Locks partitioned with the data
 - Independent: each node manages “its own” lock table
 - Works for objects that fit on one node (pages, tuples)
- For coarser-grained locks, assign a “home” node
 - Object being locked (table, DB) exists across nodes
 - These locks can be partitioned across nodes
 - Or centralized at a master node



Ignore global locks for a moment...

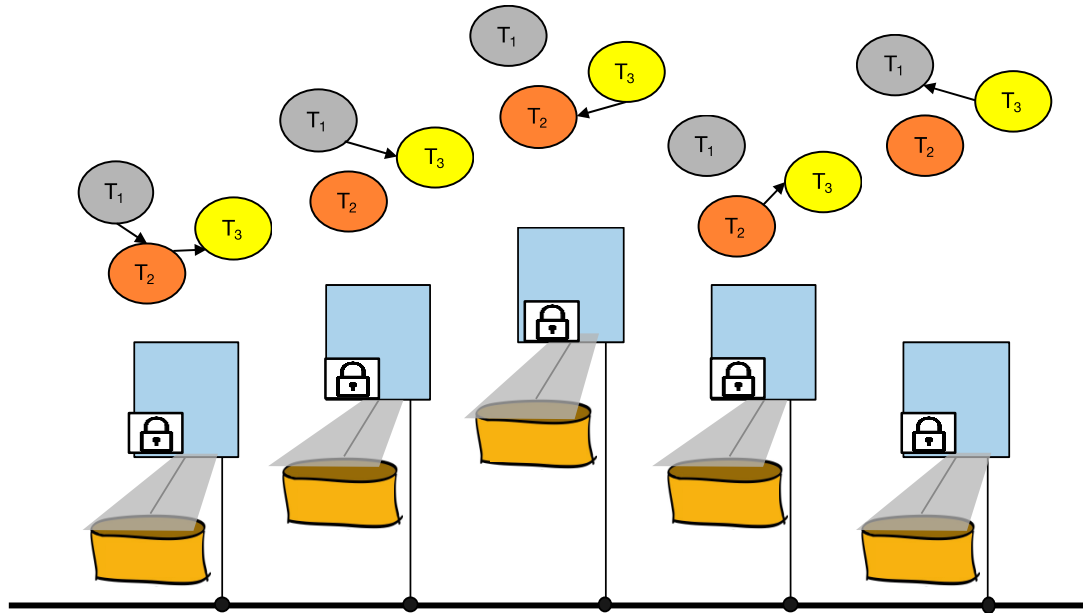
- Every node does its own locking
 - Clean and efficient
- “Global” issues remain:
 - Deadlock
 - Commit/Abort



DISTRIBUTED DEADLOCK DETECTION

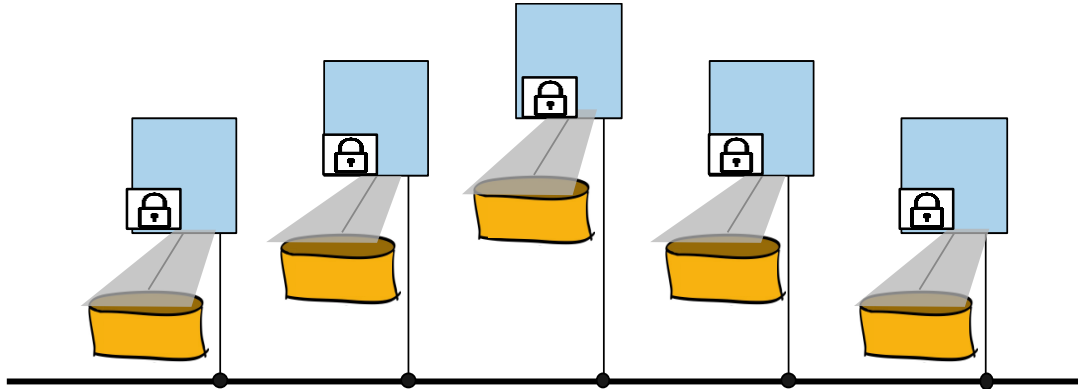
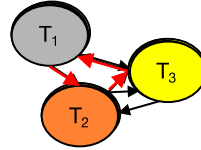
What Could Go Wrong? #1

- Deadlock detection



What Could Go Wrong? #1 Part 2

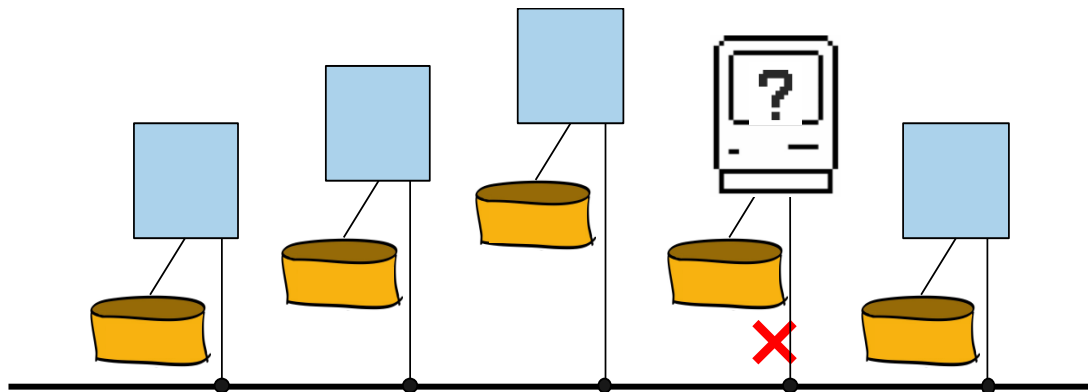
- Deadlock detection
 - Easy fix: periodically union at designated master



DISTRIBUTED COMMIT: 2PC

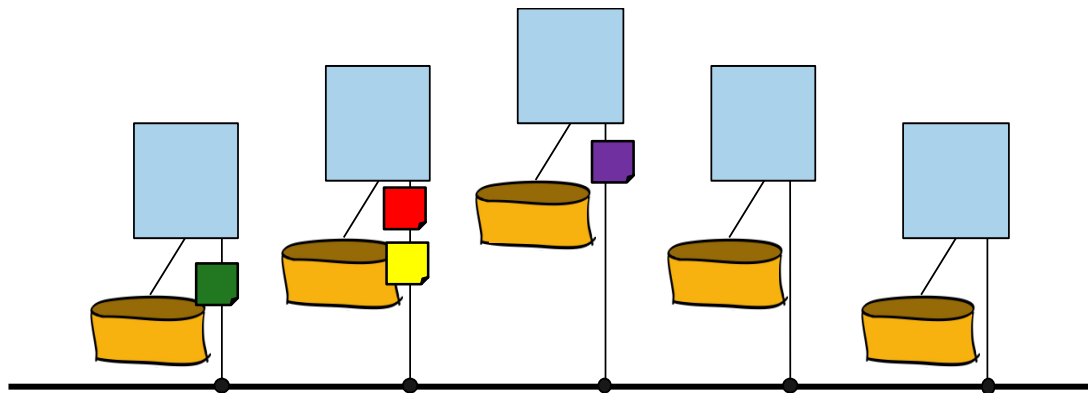
What Could Go Wrong? #2

- Failures/Delays: Nodes
 - Commit? Abort?
 - When the node comes back, how does it recover in a world that moved forward?



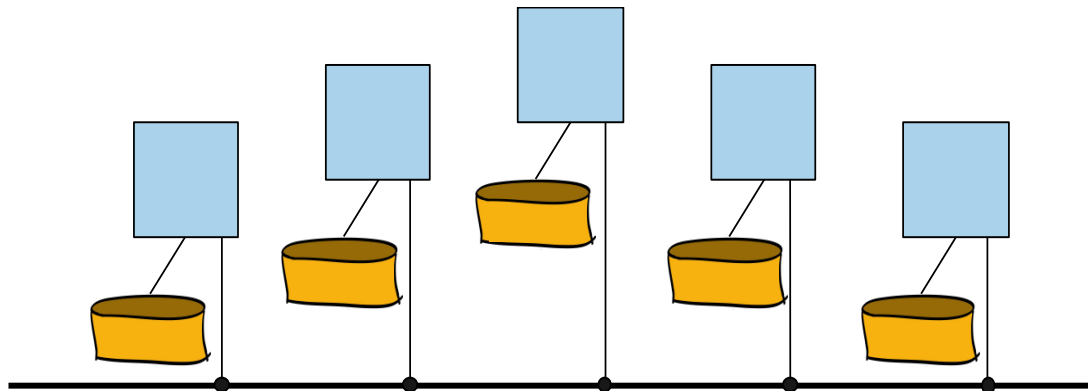
What Could Go Wrong? #2, Part 2

- Failures/Delays: Nodes
- Failures/Delays: Messages
 - Non-deterministic reordering per channel, interleaving across channels
 - “Lost” (very delayed) messages



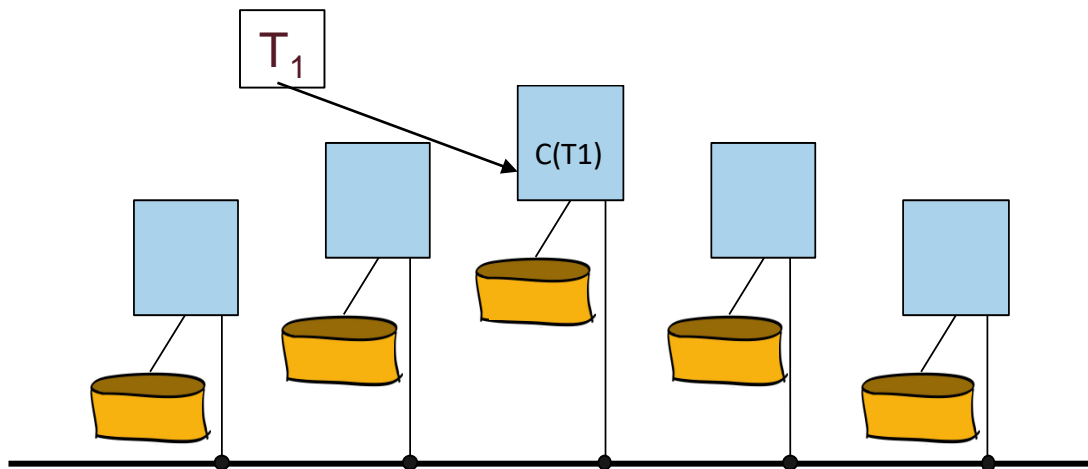
What Could Go Wrong? #2, Part 3

- Failures/Delays: Nodes
- Failures/Delays: Messages
 - Non-deterministic reordering per channel, interleaving across channels
 - “Lost” (very delayed) messages
- How do all nodes agree on Commit vs. Abort?



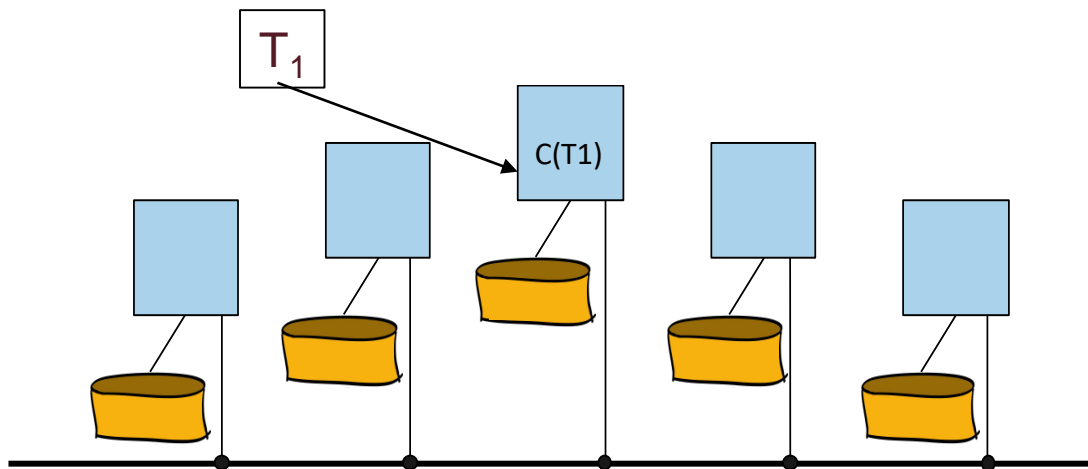
Basic Idea: Distributed Voting

- Vote for Commitment
 - How many votes does a commit need to win?
 - Any single node could observe a problem (e.g. deadlock, constraint violation)
 - Hence must be unanimous.



Distributed voting? How?

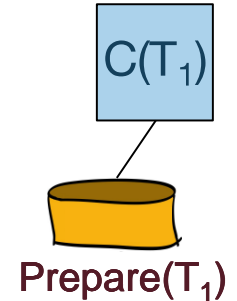
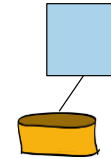
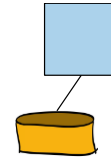
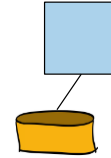
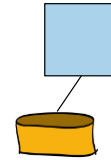
- How do we implement distributed voting?!
 - In the face of message/node failure/delay?



2-Phase Commit

- A.k.a. 2PC. (Not to be confused with 2PL!)
- Like a wedding ceremony!
- **Phase 1: “do you take this man/woman...”**
 - Coordinator tells participants to “prepare”
 - Participants respond with yes/no votes
 - Unanimity required for yes!
- **Phase 2: “I now pronounce you...”**
 - Coordinator disseminates result of the vote
- Need to do some logging for failure handling....

2-Phase Commit, Part 1



- **Phase 1:**

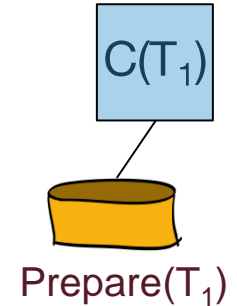
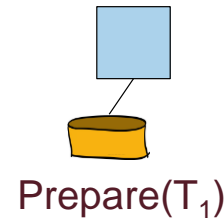
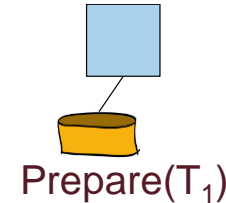
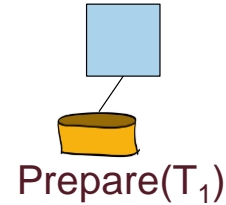
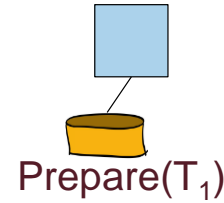
- **Coordinator tells participants to “prepare”**
- Participants respond with yes/no votes
 - Unanimity required for commit!

- **Phase 2:**

- Coordinator disseminates result of the vote
- Participants respond with Ack

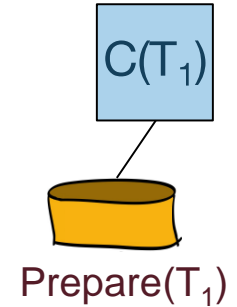
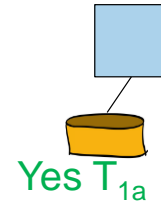
2-Phase Commit, Part 2

- **Phase 1:**
 - **Coordinator tells participants to “prepare”**
 - Participants respond with yes/no votes
 - Unanimity required for commit!
- **Phase 2:**
 - Coordinator disseminates result of the vote
 - Participants respond with Ack

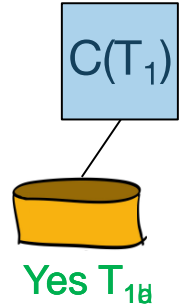
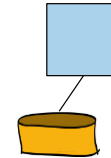
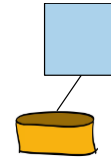
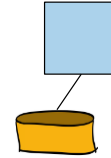
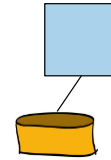


2-Phase Commit, Part 3

- **Phase 1:**
 - Coordinator tells participants to “prepare”
 - **Participants respond with yes/no votes**
 - **Unanimity required for commit!**
- **Phase 2:**
 - Coordinator disseminates result of the vote
 - Participants respond with Ack



2-Phase Commit, Part 4



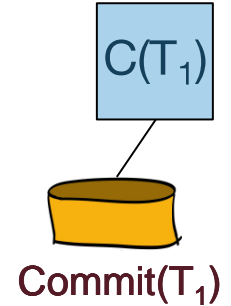
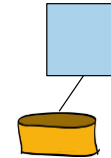
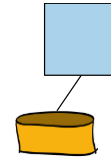
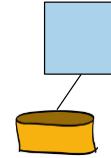
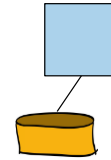
- **Phase 1:**

- Coordinator tells participants to “prepare”
- **Participants respond with yes/no votes**
 - **Unanimity required for commit!**

- **Phase 2:**

- Coordinator disseminates result of the vote
- Participants respond with Ack

2-Phase Commit, Part 5



- Phase 1:

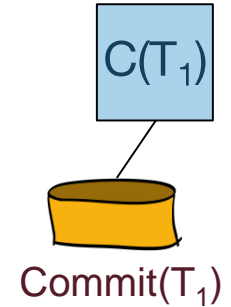
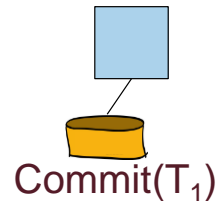
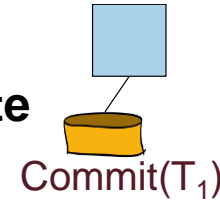
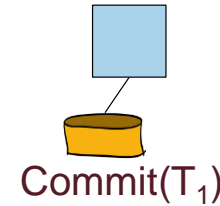
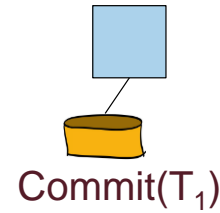
- Coordinator tells participants to “prepare”
- Participants respond with yes/no votes
 - Unanimity required for commit!

- Phase 2:

- Coordinator disseminates result of the vote**
- Participants respond with Ack

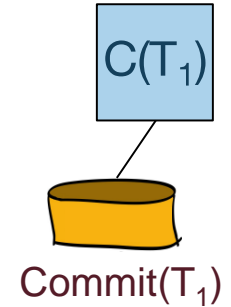
2-Phase Commit, Part 6

- Phase 1:
 - Coordinator tells participants to “prepare”
 - Participants respond with yes/no votes
 - Unanimity required for commit!
- **Phase 2:**
 - **Coordinator disseminates result of the vote**
 - Participants respond with Ack

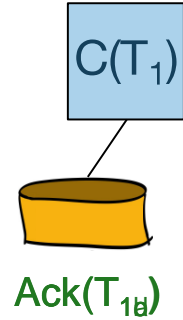
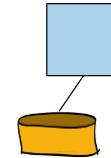
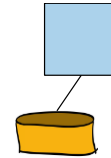
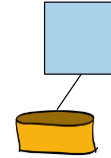
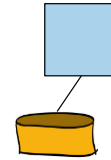


2-Phase Commit, Part 7

- Phase 1:
 - Coordinator tells participants to “prepare”
 - Participants respond with yes/no votes
 - Unanimity required for commit!
- **Phase 2:**
 - Coordinator disseminates result of the vote
 - **Participants respond with Ack**



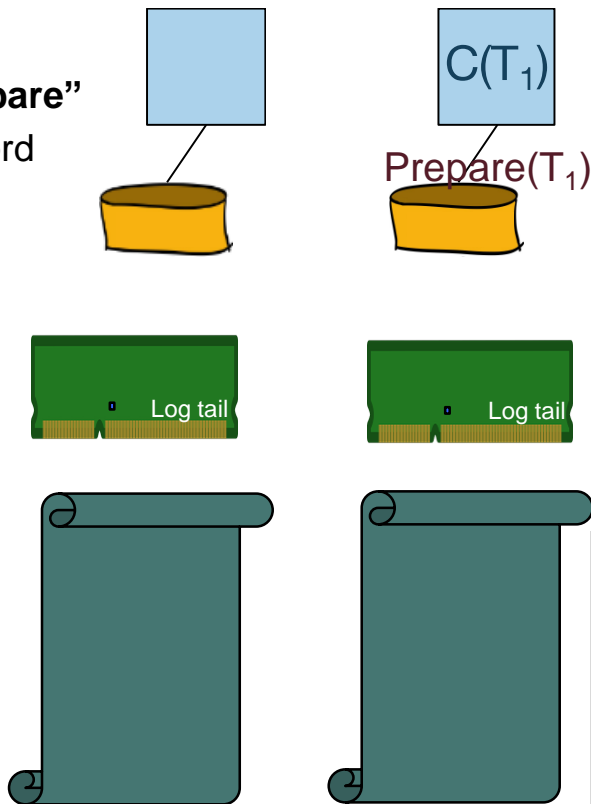
2-Phase Commit, Part 8



- Phase 1:
 - Coordinator tells participants to “prepare”
 - Participants respond with yes/no votes
 - Unanimity required for commit!
- **Phase 2:**
 - Coordinator disseminates result of the vote
 - **Participants respond with Ack**

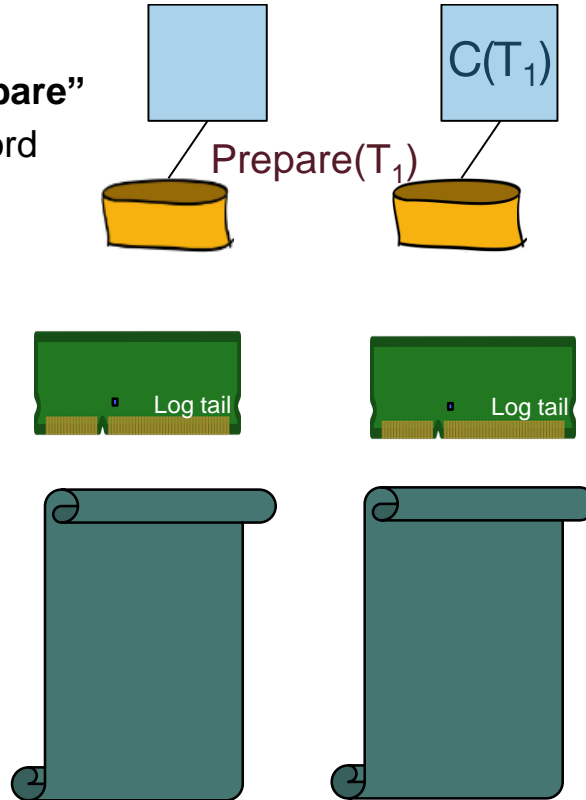
One More Time, With Logging

- **Phase 1**
- **Coordinator tells participants to “prepare”**
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



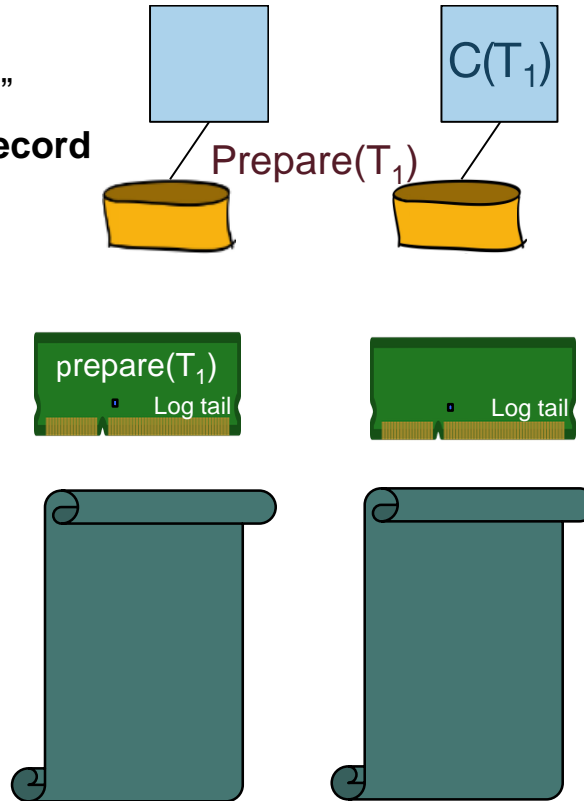
One More Time, With Logging, Part 2

- **Phase 1**
- **Coordinator tells participants to “prepare”**
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



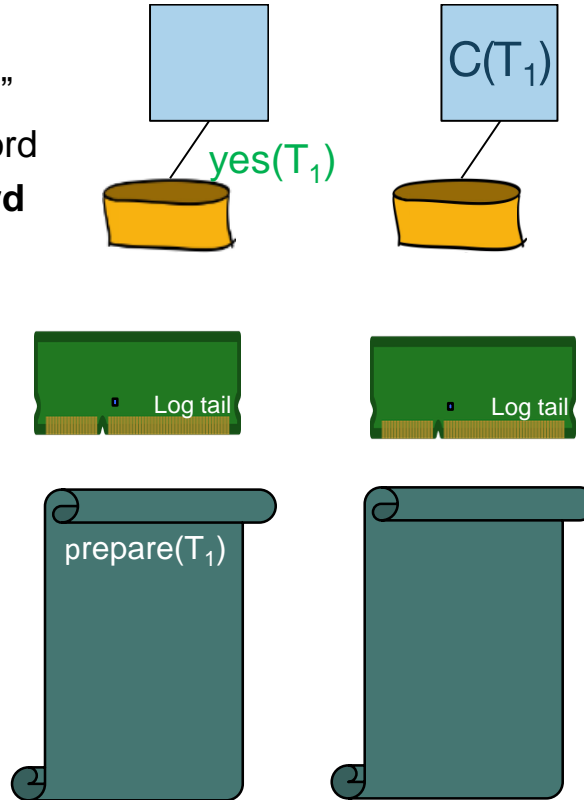
One More Time, With Logging, Part 3

- **Phase 1**
- Coordinator tells participants to “prepare”
- **Participants generate prepare/abort record**
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



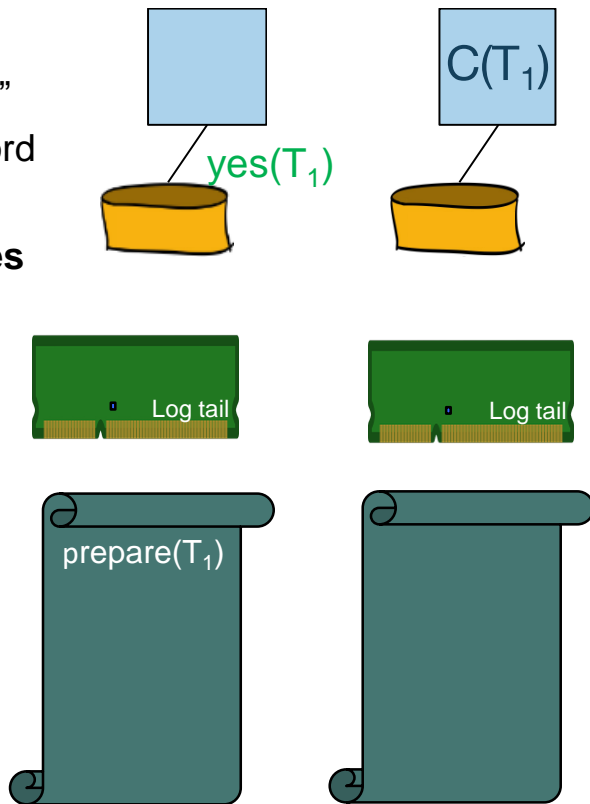
One More Time, With Logging, Part 4

- **Phase 1**
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- **Participants flush prepare/abort record**
- Participants respond with yes/no votes
- Coordinator generates commit record
- Coordinator flushes commit record



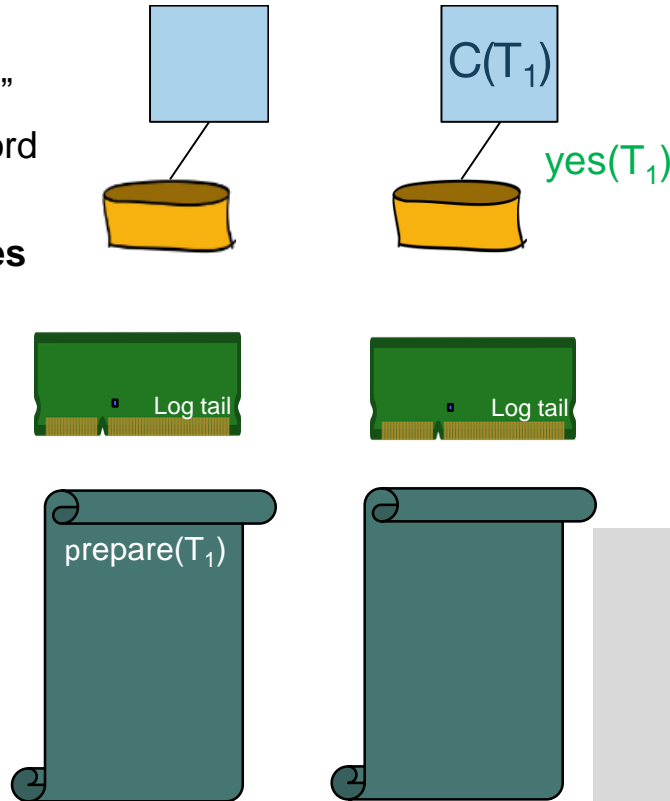
One More Time, With Logging, Part 5

- **Phase 1**
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- **Participants respond with yes/no votes**
- Coordinator generates commit record
- Coordinator flushes commit record



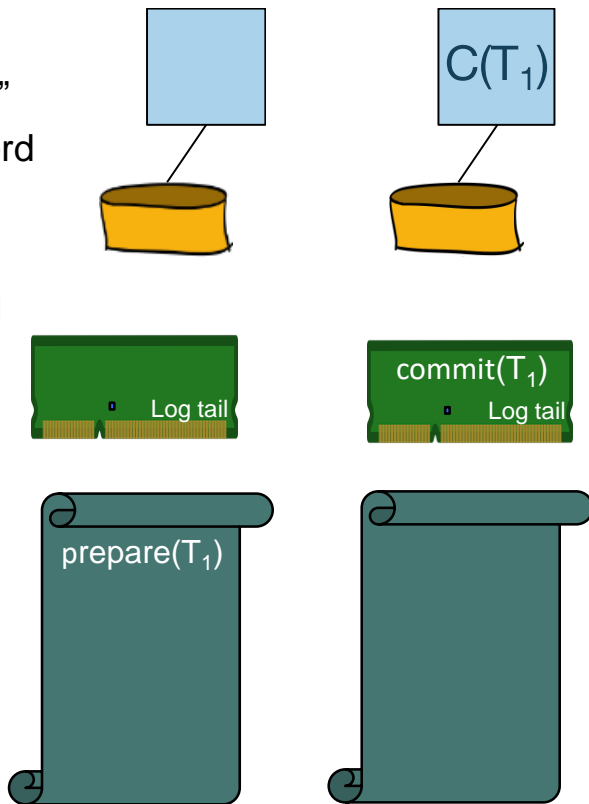
One More Time, With Logging, Part 6

- **Phase 1**
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- **Participants respond with yes/no votes**
- Coordinator generates commit record
- Coordinator flushes commit record



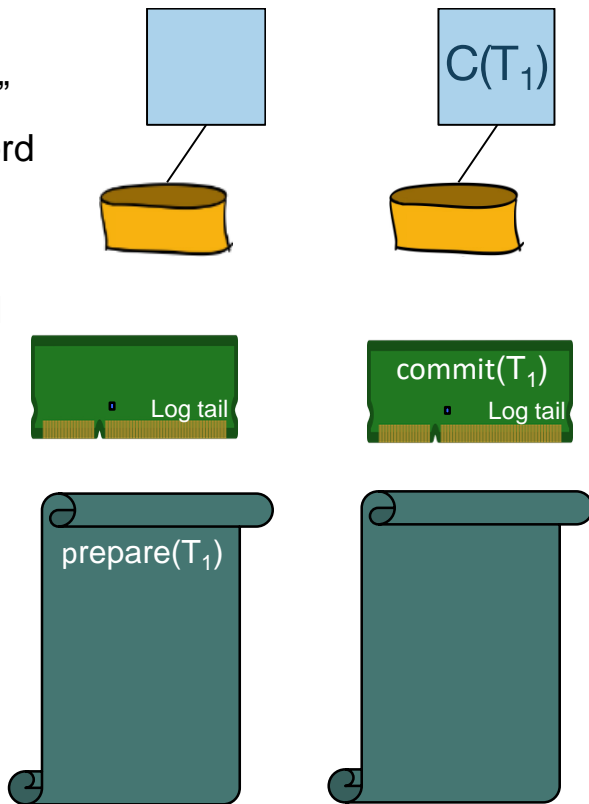
One More Time, With Logging, Part 7

- **Phase 1**
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- **Coordinator generates commit record**
- Coordinator flushes commit record



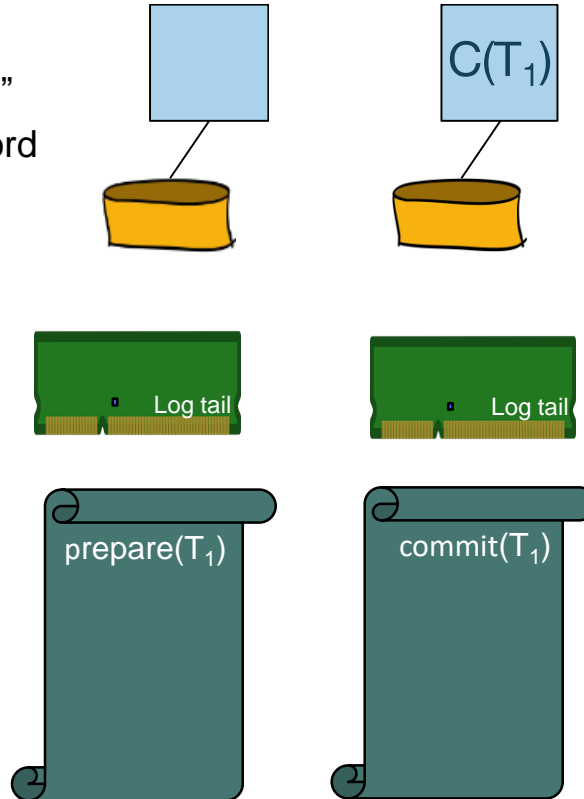
One More Time, With Logging, Part 8

- **Phase 1**
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- **Coordinator generates commit record**
- Coordinator flushes commit record



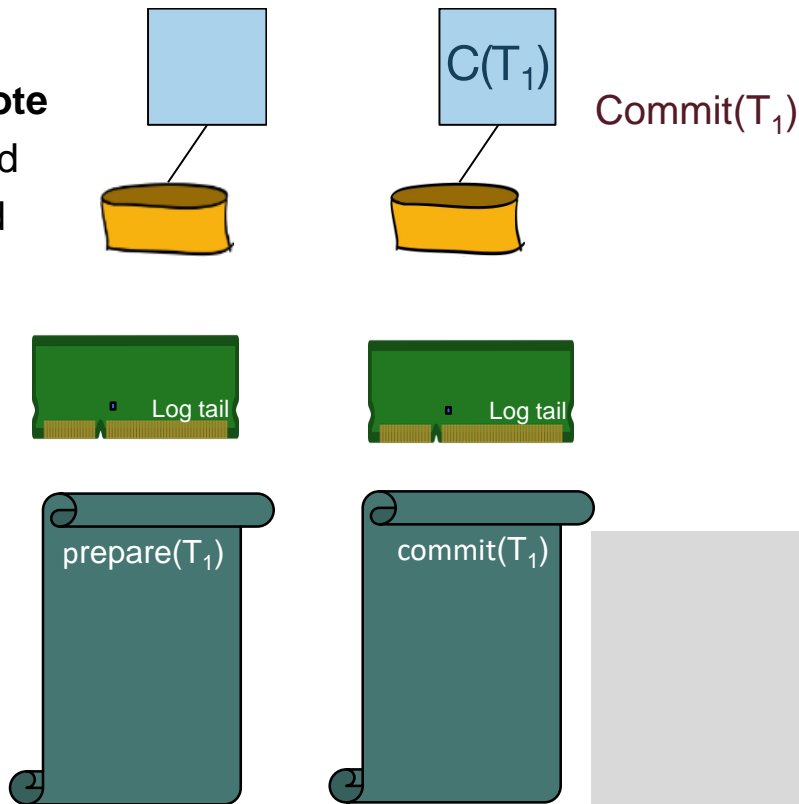
One More Time, With Logging, Part 9

- **Phase 1**
- Coordinator tells participants to “prepare”
- Participants generate prepare/abort record
- Participants flush prepare/abort record
- Participants respond with yes/no votes
- Coordinator generates commit record
- **Coordinator flushes commit record**



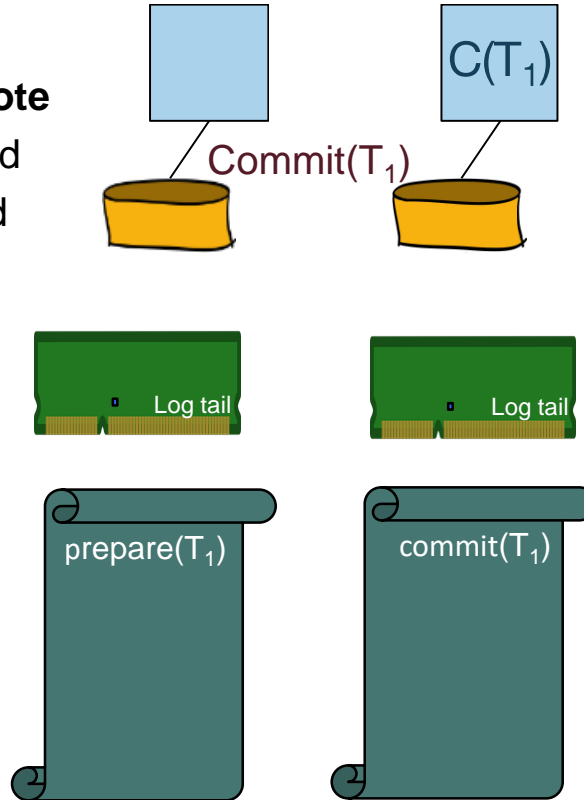
One More Time, With Logging, Part 10

- **Phase 2:**
- **Coordinator broadcasts result of vote**
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



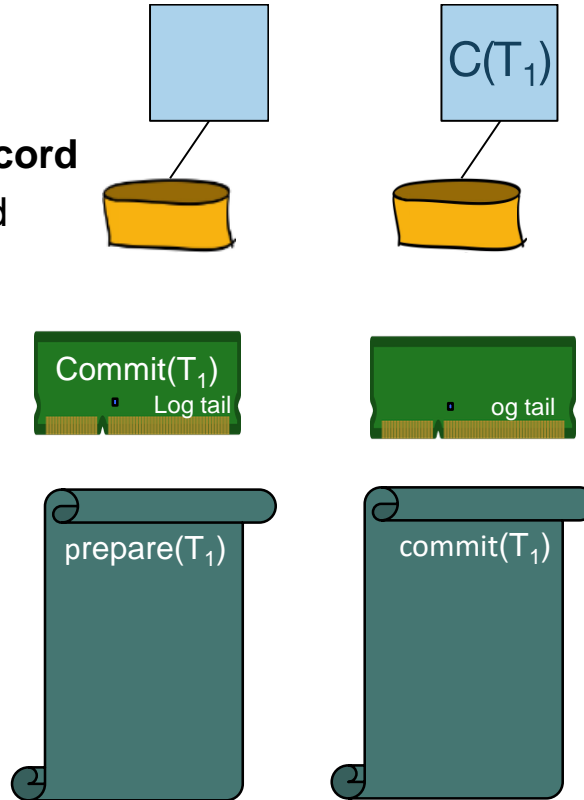
One More Time, With Logging, Part 11

- **Phase 2:**
- **Coordinator broadcasts result of vote**
- Participants make commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



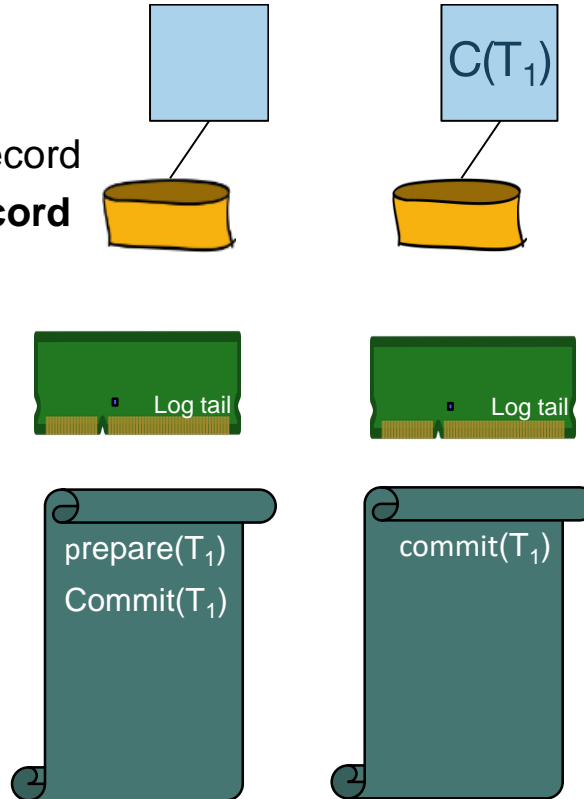
One More Time, With Logging, Part 12

- **Phase 2:**
- Coordinator broadcasts result of vote
- **Participants make commit/abort record**
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



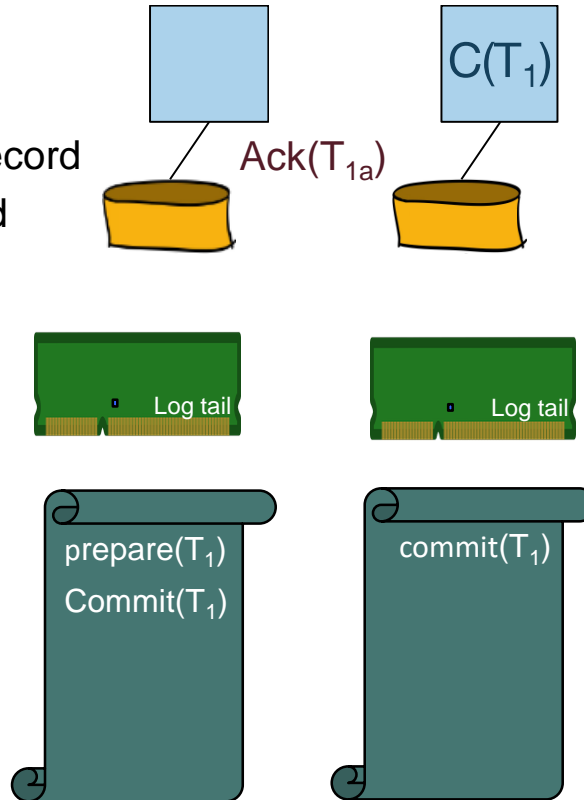
One More Time, With Logging, Part 13

- **Phase 2:**
- Coordinator broadcasts result of vote
- Participants generate commit/abort record
- **Participants flush commit/abort record**
- Participants respond with Ack
- Coordinator generates end record
- Coordinator flushes end record



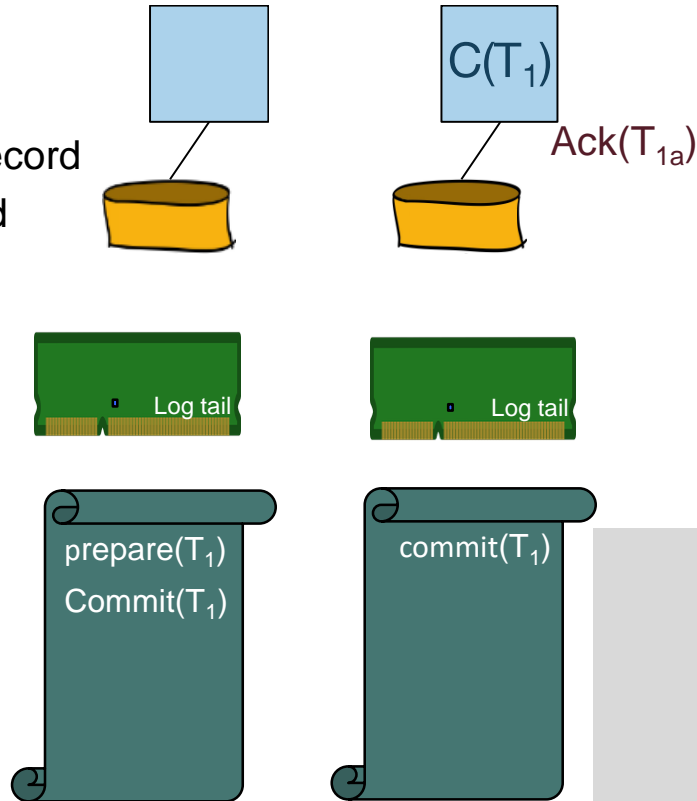
One More Time, With Logging, Part 14

- **Phase 2:**
- Coordinator broadcasts result of vote
- Participants generate commit/abort record
- Participants flush commit/abort record
- **Participants respond with Ack**
- Coordinator generates end record
- Coordinator flushes end record



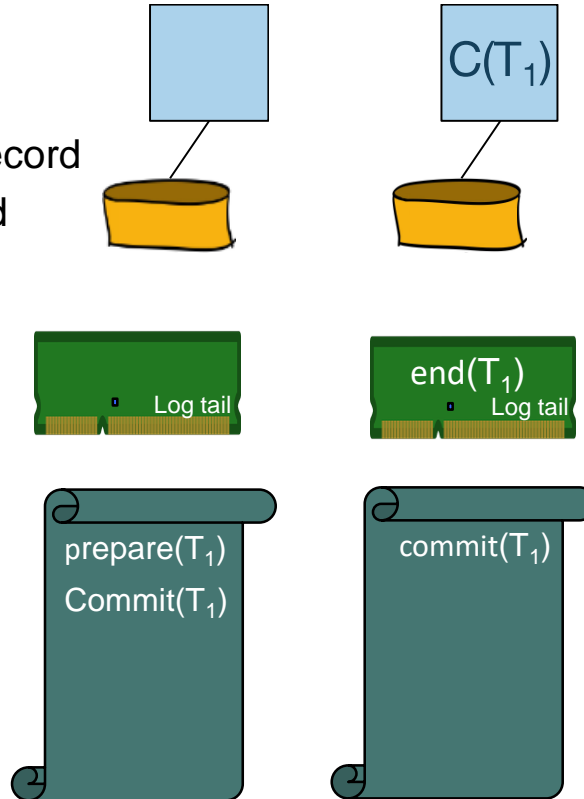
One More Time, With Logging, Part 15

- **Phase 2:**
- Coordinator broadcasts result of vote
- Participants generate commit/abort record
- Participants flush commit/abort record
- **Participants respond with Ack**
- Coordinator generates end record
- Coordinator flushes end record



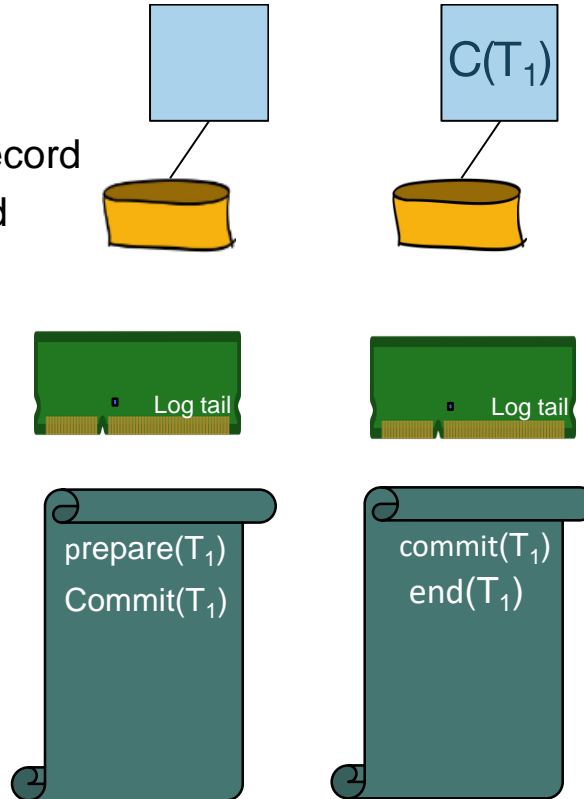
One More Time, With Logging, Part 16

- **Phase 2:**
- Coordinator broadcasts result of vote
- Participants generate commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- **Coordinator generates end record**
- Coordinator flushes end record



One More Time, With Logging, Part 17

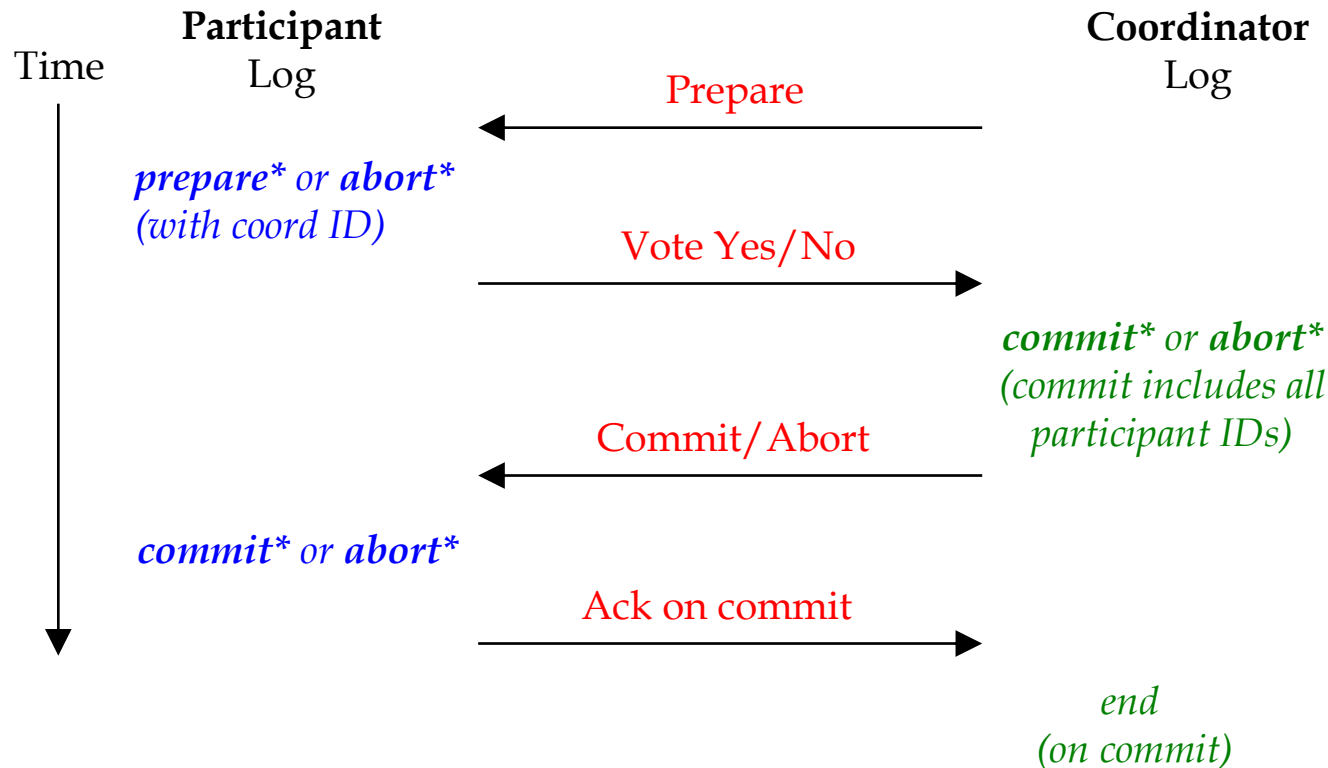
- **Phase 2:**
- Coordinator broadcasts result of vote
- Participants generate commit/abort record
- Participants flush commit/abort record
- Participants respond with Ack
- Coordinator generates end record
- **Coordinator flushes end record**



2PC In a Nutshell

NOTE

*asterisk**: wait for log flush
before sending next msg



RECOVERY AND 2PC

Failure Handling

- Assume everybody recovers eventually
 - Big assumption!
 - Depends on WAL (and short downtimes)
- Coordinator notices a Participant is down?
 - If participant hasn't voted yet, coordinate aborts transaction
 - If waiting for a commit Ack, hand to "recovery process"
- Participant notices Coordinator is down?
 - If it hasn't yet logged prepare, then abort unilaterally
 - If it has logged prepare, hand to "recovery process"
- Note
 - Thinking a node is "down" may be incorrect!

Integration with ARIES Recovery

- On recovery
 - Assume there's a "Recovery Process" at each node
 - It will be given tasks to do by the Analysis phase of ARIES
 - These tasks can run in the background (asynchronously)
- Note: multiple roles on a single node
 - Coordinator for some xacts, Participant for others

Integration with ARIES: Analysis

- Recall transaction table states
 - Running, Committing, Aborting
- On seeing Prepare log record (participant)
 - Change state to committing
 - Tell recovery process to ask coordinator recovery process for status
 - When coordinator responds, recovery process handles commit/abort as usual
 - (Note: During REDO, Strict 2PL locks will be acquired)

Integration with ARIES: Analysis, cont

- On seeing Commit/Abort log record (coordinator)
 - Change state to committing/aborting respectively
 - Tell recovery process to send commit/abort msgs to participants
 - Once all participants ack commit, recovery process writes End and forgets
- If at end of analysis there's no 2PC log records for xact X
 - Simply set to Aborting locally, and let ToUndo handle it.
 - Same for participant and coordinator
 - A.k.a. "Presumed Abort"
 - There is an optimization called "Presumed Commit"

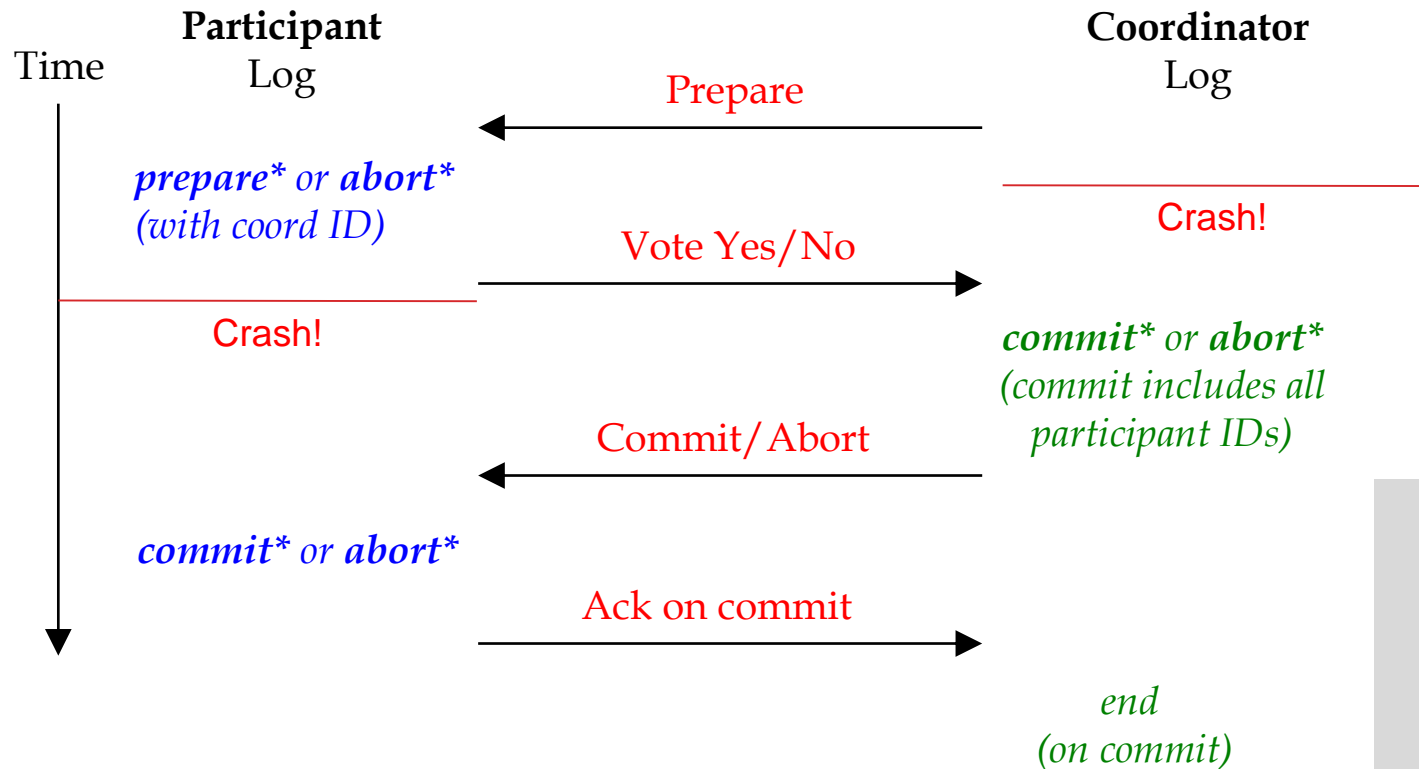
How Does Recovery Process Work?

- Coordinator recovery process gets inquiry from a “prepared” participant
 - If transaction table at coordinator says aborting/committing
 - send appropriate response and continue protocol on both sides
 - If transaction table at coordinator says nothing: send ABORT
 - Only happens if coordinator had also crashed before writing commit/abort
 - Inquirer does the abort on its end

2PC In a Nutshell

NOTE

*asterisk**: wait for log flush
before sending next msg



Recovery: Think it through

- What happens when coordinator recovers?
 - With “commit” and “end”?
 - With just “commit”?
 - With “abort”?
- What happens when participant recovers:
 - With no prepare/commit/abort?
 - With “prepare” and “commit”?
 - With just “prepare”?
 - With “abort”?

*Commit iff coordinator
logged a commit*

Recovery: Think it through, cont

- What happens when coordinator recovers?
 - With “commit” and “end”? **Nothing**
 - With just “commit”? **Rerun Phase 2!**
 - With “abort”? **Nothing (Presumed Abort)**
- What happens when participant recovers:
 - With no prepare/commit/abort? **Nothing (Presumed Abort)**
 - With “prepare” & “commit”? **Send Ack to coordinator.**
 - With just “prepare”? **Send inquiry to Coordinator**
 - With “abort”? **Nothing (Presumed Abort)**

*Commit iff coordinator
logged a commit*

2PC + 2PL

- Ensure point-to-point messages are densely ordered
 - 1,2,3,4,5...
 - Dense per (sender/receiver/XID)
 - Receiver can detect anything missing or out-of-order
 - Receiver buffers message $k+1$ until $[1..k]$ received
- Commit:
 - When a participant processes Commit request, it has all the locks it needs
 - Flush log records and drop locks atomically
- Abort:
 - Its safe to abort autonomously, locally: no cascade.
 - Log appropriately to 2PC (presumed abort in our case)
 - Perform local Undo, drop locks atomically

Availability Concerns

- What happens while a node is down?
 - Other nodes may be in limbo, holding locks
 - So certain data is unavailable
 - This may be bad...
- Dead Participants? Respawned by coordinator
 - Recover from log
 - And if the old participant comes back from the dead, just ignore it and tell it to recycle itself
- Dead Coordinator?
 - This is a problem!
 - 3-Phase Commit was an early attempt to solve it
 - Paxos Commit provides a more comprehensive solution
 - Gray+Lamport paper! Out of scope for this class.

Summing Up

- Distributed Databases
 - A central aspect of Distributed Systems
- Partitioning provides Scale-Up
- Can also partition lock tables and logs
- But need to do some global coordination:
 - Deadlock detection: easy
 - Commit: trickier
- Two-phase commit is a classic distributed consensus protocol
 - Logging/recovery aspects unique:
 - many distributed protocols gloss over
 - But 2PC is unavailable on any single failure
 - This is bad news for scale-up,
 - because odds of failure go up with #machines
 - Paxos Commit (Gray+Lamport) addresses that problem