# Distributed Transactions and Atomic Commit

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

# Overview

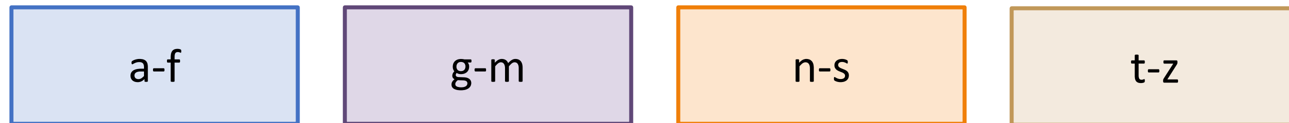
- Introduction to distributed transactions
- Atomic commit

# Review of transactions

- Transactions may read and write multiple items
- Transactional systems ensure atomic execution
  - Use concurrency control for isolation (hide concurrency)
    - Read and write accesses of a transaction appear to execute together
  - Use logging for failure atomicity and durability (hide failures)
    - Allows partial modifications to be rolled back (for failure atomicity), and completed modifications to be rolled forward (for durability)
- Until now, assumed transactions access items on one node

# Sharding

- With increasing data volume, data needs to be stored on multiple nodes
- Typically, data is partitioned (or sharded) across nodes



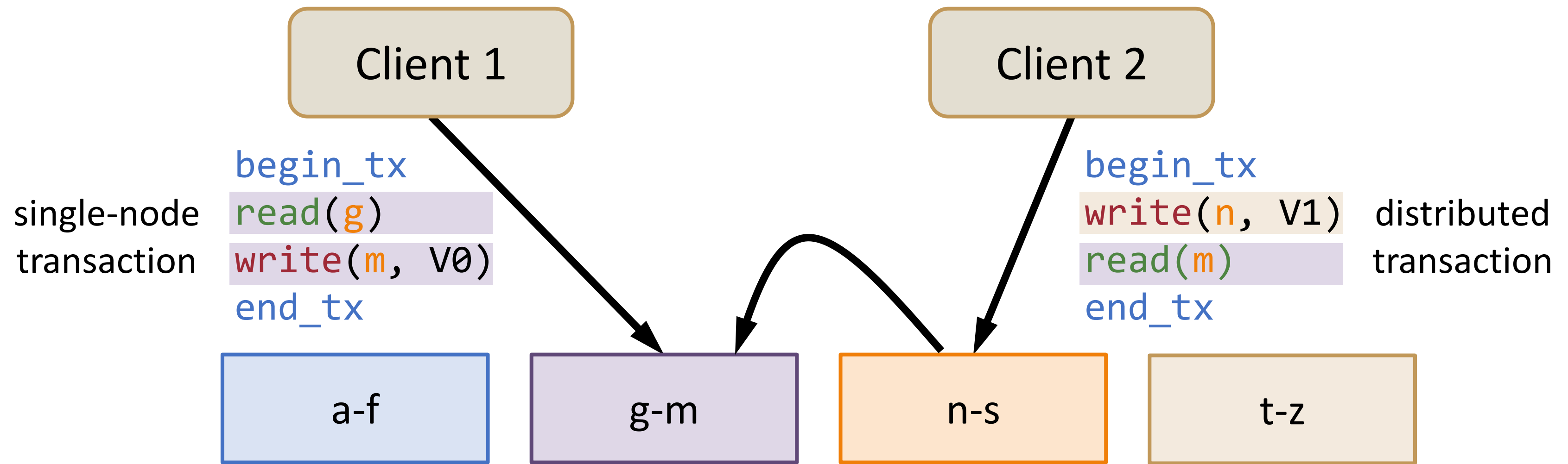

- Benefits of sharding
  - Partitioning data across storage nodes helps scale performance via parallelization and load balancing
  - Data can be located close to users for better latency

# Distributed transactions

- Transactions that access items stored on multiple nodes are called distributed transactions
  - Data is accessed and updated on all nodes in ACID manner
  - Physically partitioned storage nodes behave like a single logical storage system
- Main challenge with distributed transactions is ensuring atomicity and durability across nodes

# Single node vs distributed transactions

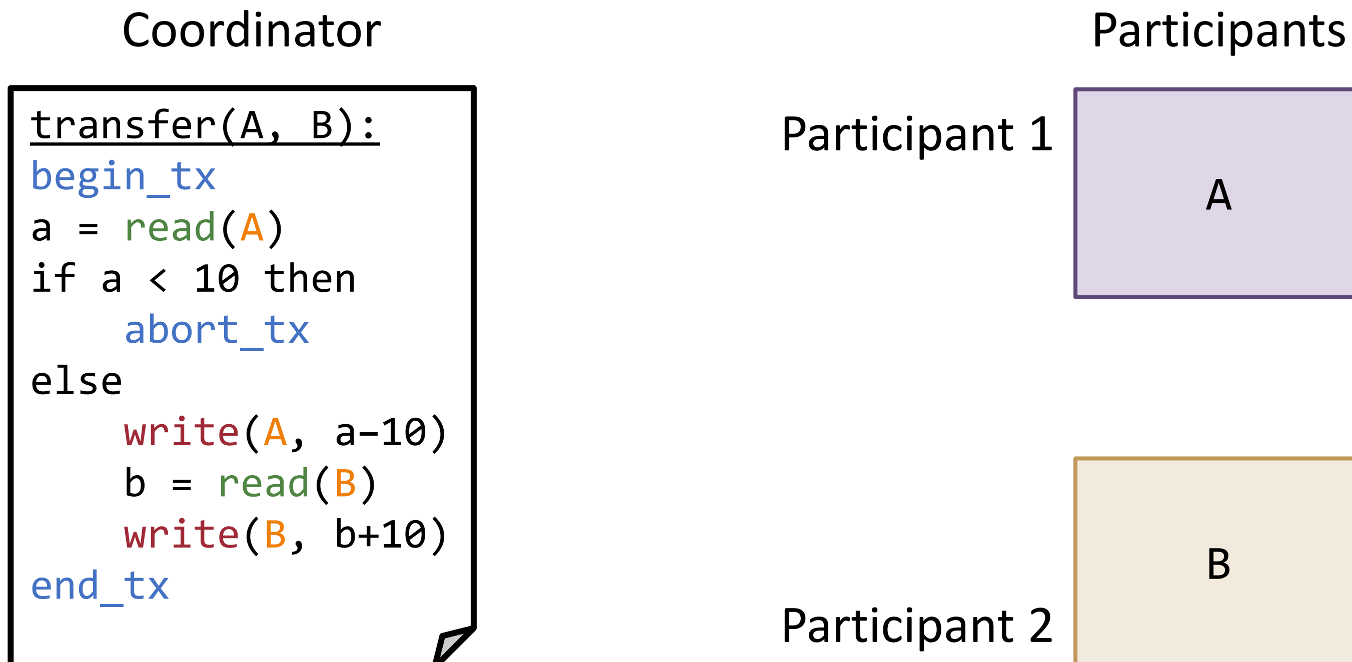
- Assume, clients send transactions to one of the nodes




- Single-node transactions access items from one node
- Distributed transactions access items from multiple nodes

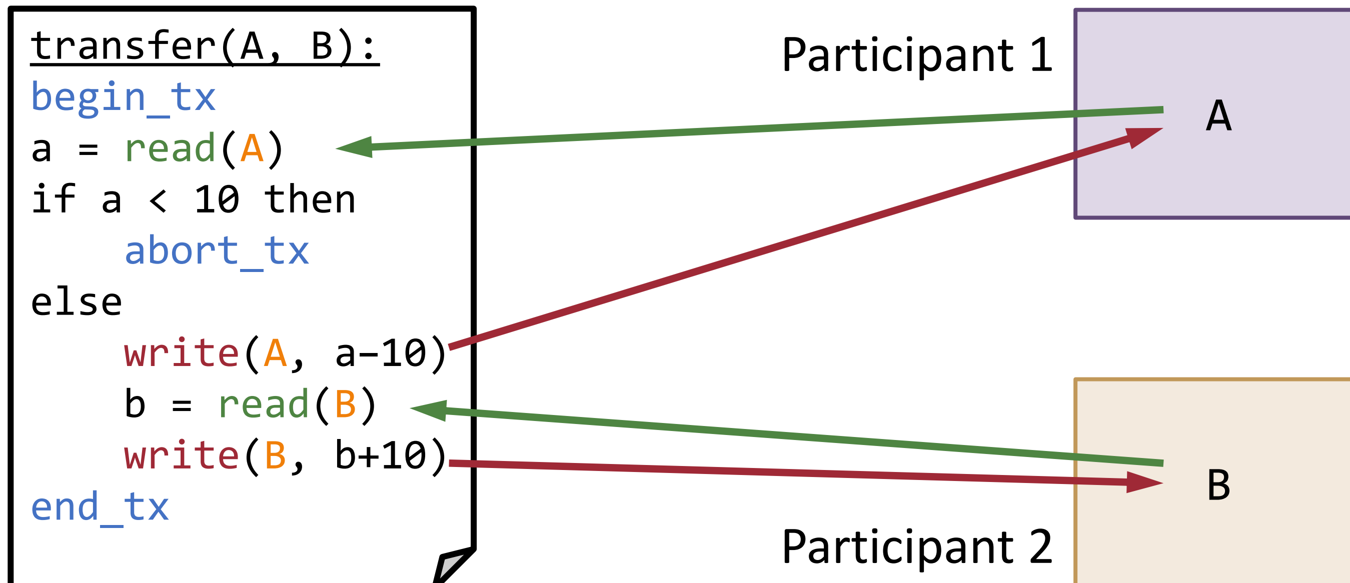
# Distributed transaction execution model

- Coordinator node receives and runs transaction code, participants nodes store data items
- Typically, coordinator is a participant as well



# Distributed transaction execution model

- Coordinator sends read/write RPC requests to participants



# Distributed transaction execution model

- Coordinators
  - Concurrent transactions may have different coordinators
- Transaction ID
  - Coordinator assigns a unique ID (TID) to each transaction
  - RPC messages, transaction state at nodes are tagged with TID
- Participants
  - Acquire locks when data item is accessed (2PL), or at commit (OCC), and wait if item is locked
  - Log modifications and install them on commit
  - Release locks on commit

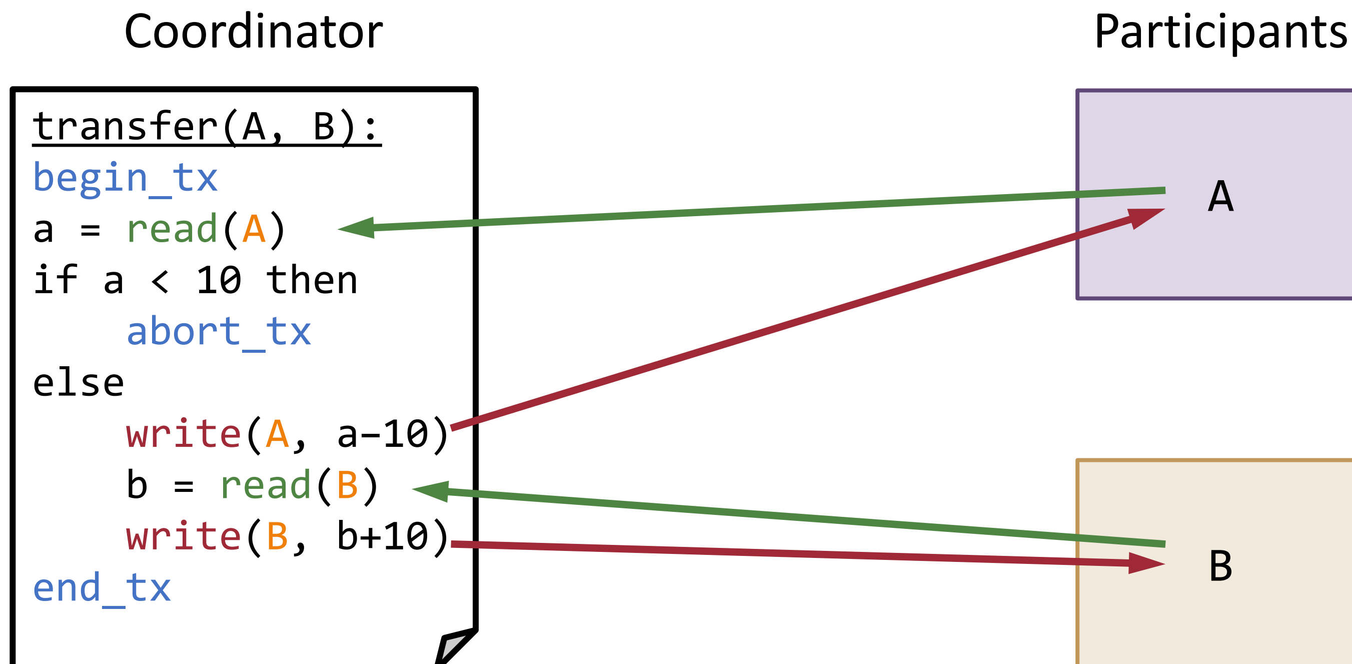
# Atomic Commit

# Atomic commit

- Problems with distributed transactions
  - One participant performs all accesses successfully, another crashes
  - One participant performs all accesses successfully, another aborts
    - Transaction constraint fails (e.g., a < 10)
    - Cannot acquire required lock (e.g., deadlock)
    - No memory or disk space available to perform read/write
  - Both participants perform all accesses, but aren't sure about other
    - Recall Two Generals problem!
- We need atomic commit
  - All nodes agree to execute transaction (commit), or else
  - Even if one node fails in any way, no node does anything (abort)
- Key idea: use a mediator node to achieve agreement

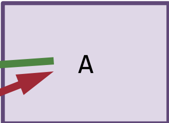
# Two-phase commit

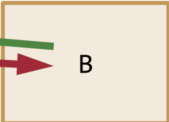
- A protocol for ensuring atomic commit
- Protocol runs after transaction execution is done
- Coordinator node serves as mediator



# Two-phase commit protocol

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
    abort_tx
else
    write(A, a-10)
    b = read(B)
    write(B, b+10)
end_tx
```



# Two-phase commit

- Phase 1: vote collection
  - Coordinator sends PREPARE message to all participants
  - Each participant votes yes or no
    - Records vote, locks held, in its log (in addition to logged updates)
  - Each participant sends yes or no VOTE response to coordinator
  - Coordinator inspects all votes
    - If all yes, then commit, else abort transactions
    - Records Commit/Abort decision in log (commit point)
    - Responds to client
- Phase 2: send decision
  - Coordinator sends DECISION message to all participants
  - Each participant commits or aborts changes, releases locks, sends ACK response to the coordinator

# Two-phase commit guarantees

- Under no failures, easy to see that 2PC guarantees:
  - Atomic commit
    - Participants commit when all prepared to commit, or else all abort
  - Durability
    - After coordinator commits, participants will apply changes
- But what happens under failures?

# Types of failures

- A participant (PA or PB) or transaction coordinator (TC) can
  - Crash and restart
  - Time out waiting for a message
    - Node is up, but didn’t receive expected message
    - Maybe the other node crashed, maybe network has failed
    - We usually can’t tell the difference, so must be correct in either case

# Participant crash failures

- What if PA crashes:
  - Before logging vote
    - PA hasn't sent VOTE to TC
    - TC could not have decided commit
    - On reboot, PA can abort and forget transaction
  - After logging NO vote
    - TC could not have decided commit
    - On reboot, PA can abort and forget transaction
  - After logging YES vote
    - TC may decide to commit
    - On reboot, PA should reacquire locks, wait for TC to send DECISION
  - After receiving DECISION
    - On reboot, PA should reacquire locks, wait for TC to resend DECISION

# Coordinator crash failures

- What if TC crashes:
  - Before logging decision
    - TC hasn't sent DECISION
    - On reboot, TC can decide to abort transaction and send DECISION
  - After logging decision
    - Some participants may have received decision, others not
    - On reboot, TC must send (same) DECISION

# Time out failures

- What if Participant PA times out waiting for PREPARE:
  - TC could not have decided commit
  - PA can abort transaction
  - Respond No to later PREPARE message
- What if TC times out waiting for VOTE from PA:
  - TC could not have sent DECISION yet
  - TC can decide to abort transaction and send DECISION
- What if PA voted YES, times out waiting for DECISION:
  - Can’t abort, since TC could have decided Commit and let PB know
  - Can’t commit, since TC could have decided Abort
  - PA must keep waiting for TC’s DECISION forever!

# Forgetting transaction state

- When can PA forget about a committed transaction?
  - After it sends ACK
  - If it gets another DECISION,
    and has no record of the transaction, it sends ACK again
- When can TC forget about a committed transaction?
  - If it sees ACK from every participant
  - Then no participant will ever need to ask again

# Two-phase commit cost

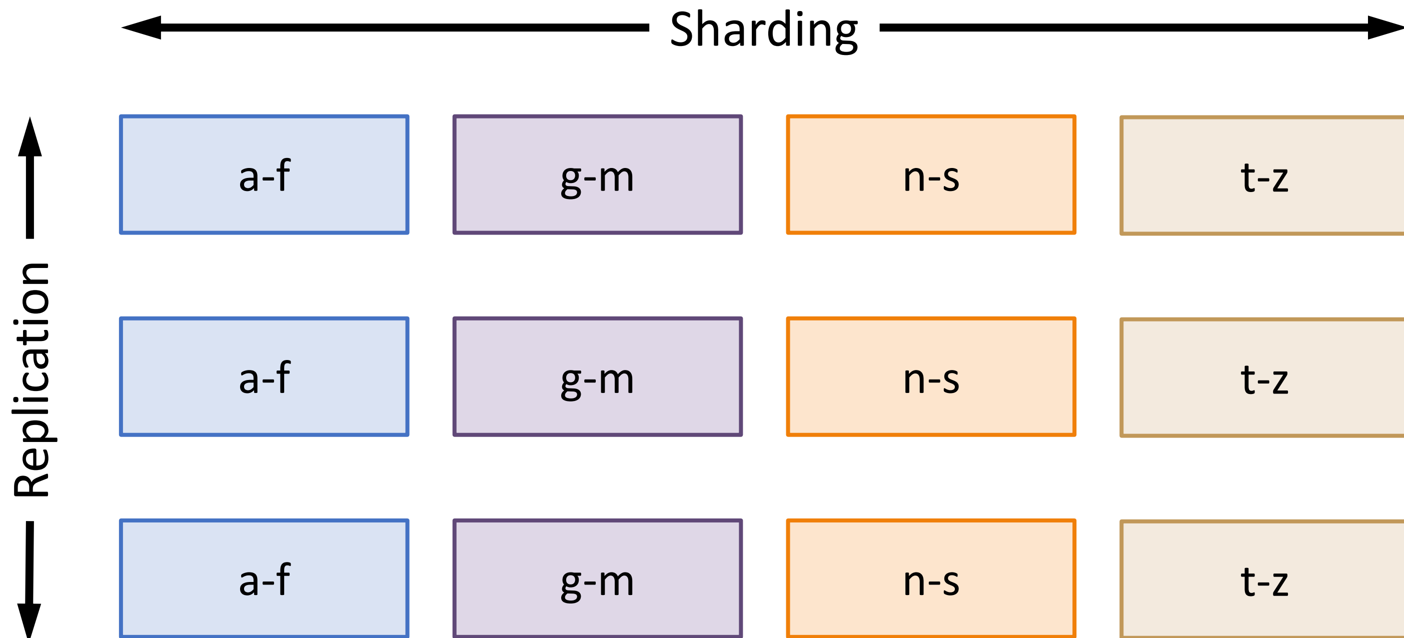
- Two-phase commit makes distributed transactions costly
  - Latency
    - Requires two additional round trips after transaction code completes
    - Votes and decision are logged to disk synchronously
  - Throughput
    - Locks are held from the time reads and writes are performed (2PC) or from prepare phase (OCC) until the end of two-phase commit
    - Other transactions waiting on locks are also delayed
  - Scalability
    - Need to handle more distributed transactions with more nodes
  - Availability
    - Coordinator crash blocks participants (while they hold locks!)

# Two-phase commit in practice

- Typically, distributed transactions used within data center
  - Round-trip times are short, network failures unlikely
- Much research on speeding up distributed transactions
  - Key idea is to limit the power of transactions
    - E.g., ensure that participants do not need to abort,
      look for "It's Time to Move on from Two Phase Commit"
    - E.g., perform transaction operations during commit,
      look for Sinfonia mini-transactions

# Distributed transactions and replication

- We have seen distributed transactions on sharded data
- How does that relate to replication?

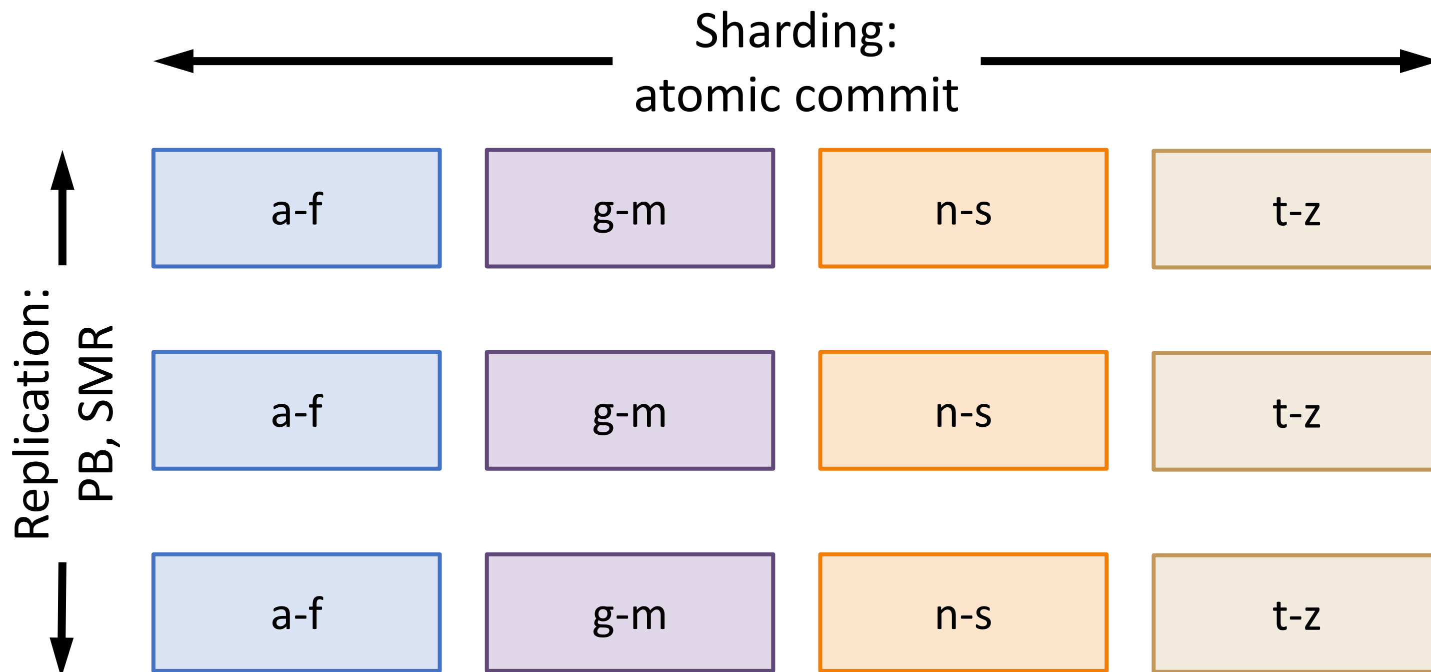


# Replication, sharding, atomic commit

- Replication is about doing same thing in multiple places
  - Can use majority consensus, since nodes store same data
  - Enables handling node failures, primarily for high availability
- Sharding is about doing different things in multiple places
  - Enables running operations concurrently, primarily for scalability
- Atomic commit is about doing different things in multiple places together (all-or-nothing atomicity)
  - Can't use majority consensus, since nodes store different data
  - A single failed node blocks progress, limits availability

# Replication, sharding, atomic commit

- Replication for fault tolerance
- Sharding for scalability, atomic commit for all-or-nothing
- Modern databases support both, e.g., Google Spanner



# Conclusions

- Transactions enable executing operations atomically
  - All accesses appear to execute together (hide concurrency)
    - Need concurrency control algorithms
  - All accesses execute or none (hide failures)
    - Need WAL and undo/redo recovery
    - Need atomic commit for distributed transactions
- Distributed transactions and atomic commit
  - Requires logging (at coordinator and participants)
  - Requires two phases, for collecting votes, and sending decision