Distributed Transactions and Atomic Commit

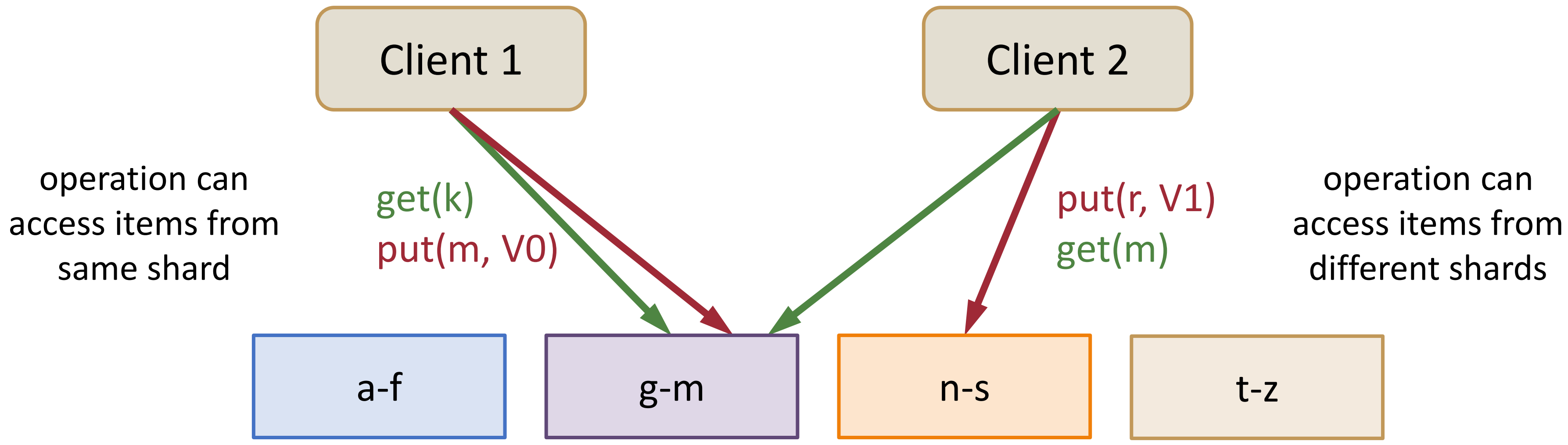
Ashvin Goel

Electrical and Computer Engineering
University of Toronto

Distributed Systems
ECE419

Motivation for transactions

- When operations access multiple items, we would like them to execute atomically
  - Appear to execute all accesses together (hide concurrency)
  - Appear to execute all accesses or none (hide failures)
- Transactions provide these semantics



Transaction guarantees: ACID

- Atomic: transaction executes completely or not at all, despite failures
- Consistent: system ensures application-specific invariants
- Isolated: no interference between concurrent transactions
- Durable: committed transactions are not lost, despite failures

Now, we will see how transactions can ensure correctness under failures

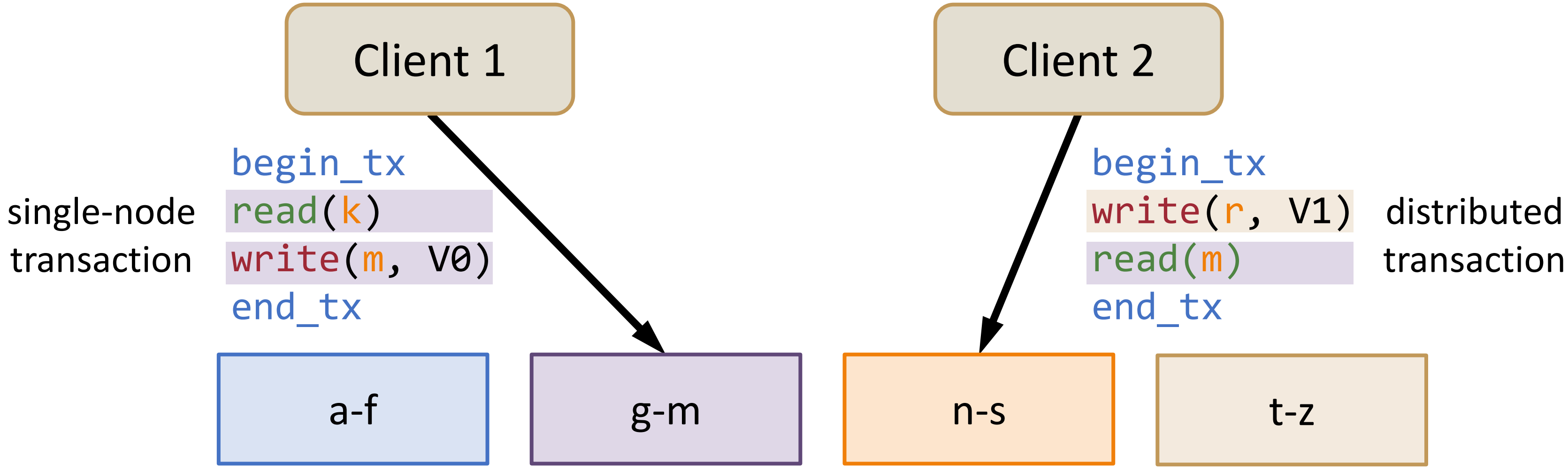
We have seen how concurrency control ensures correctness under concurrency

Ensuring atomicity and durability

- We have already looked at write-ahead logging (WAL)
  - With WAL, system logs a modified item before overwriting it
  - Allows partial modifications to be rolled back (for atomicity), and completed modifications to be rolled forward (for durability)
- Are we done?
  - When discussing write-ahead logging,
    we assumed that an operation accesses items on one node
- What if transactions access items from multiple nodes?
  - We need atomicity and durability across nodes
  - Either all nodes execute transaction and make its updates durable, or all nodes roll back any updates made by a transaction

Single node vs distributed transactions

- Assume items are sharded across nodes



- Clients send their transactions to one of the nodes
- Single-node transactions access items from one node
- Distributed transactions access items from multiple nodes

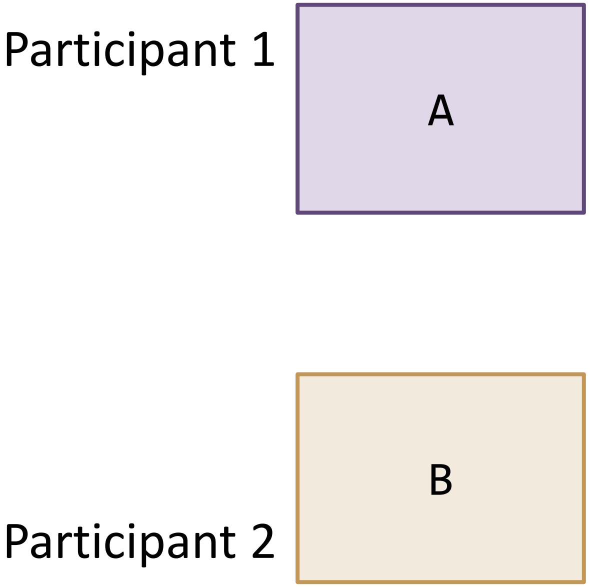
Distributed transaction execution model

- Coordinator node receives and runs transaction code, participants nodes store data records

Coordinator

```
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
```

Participants

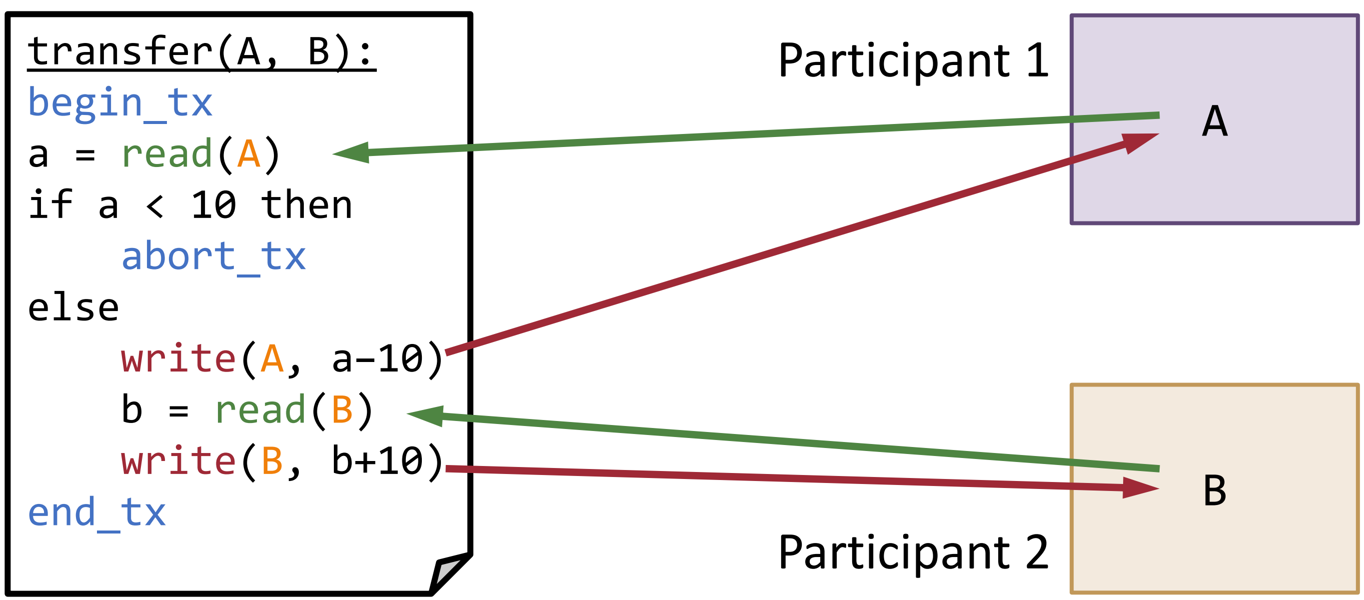


Distributed transaction execution model

- Coordinator sends read/write RPC requests to participants

Coordinator node:
runs transaction code,
coordinates with participants,
uses WAL for recovery

Participant nodes:
store data records,
acquire/release locks,
use WAL for recovery



Distributed transaction execution model

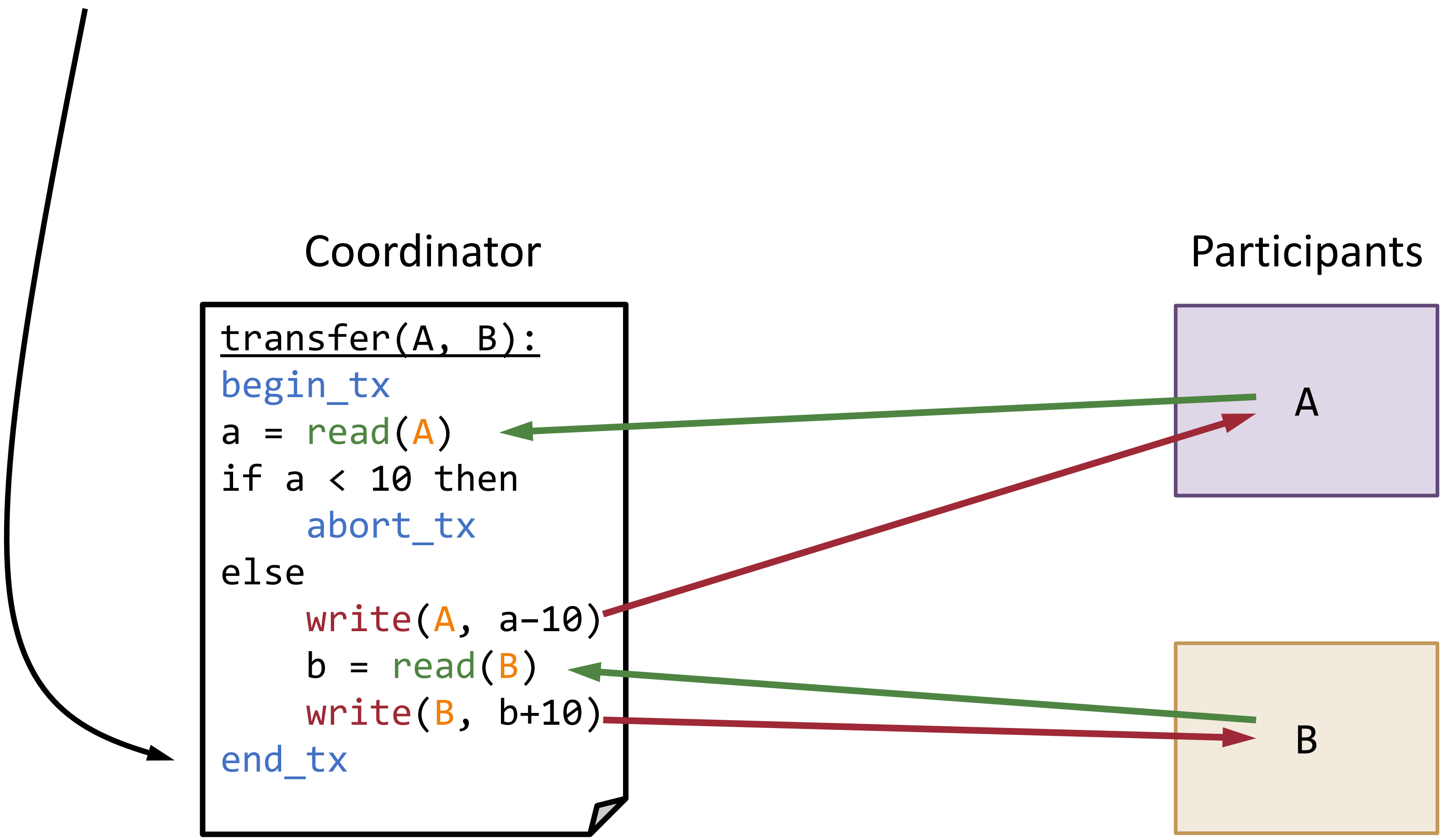
- Coordinators
  - Concurrent transactions may have different coordinators
  - A coordinator can be a participant as well
- Transaction ID
  - Coordinator assigns a unique ID (TID) to each transaction
  - RPC messages, transaction state at nodes are tagged with TID
- Participants
  - Acquire locks when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

Atomic commit

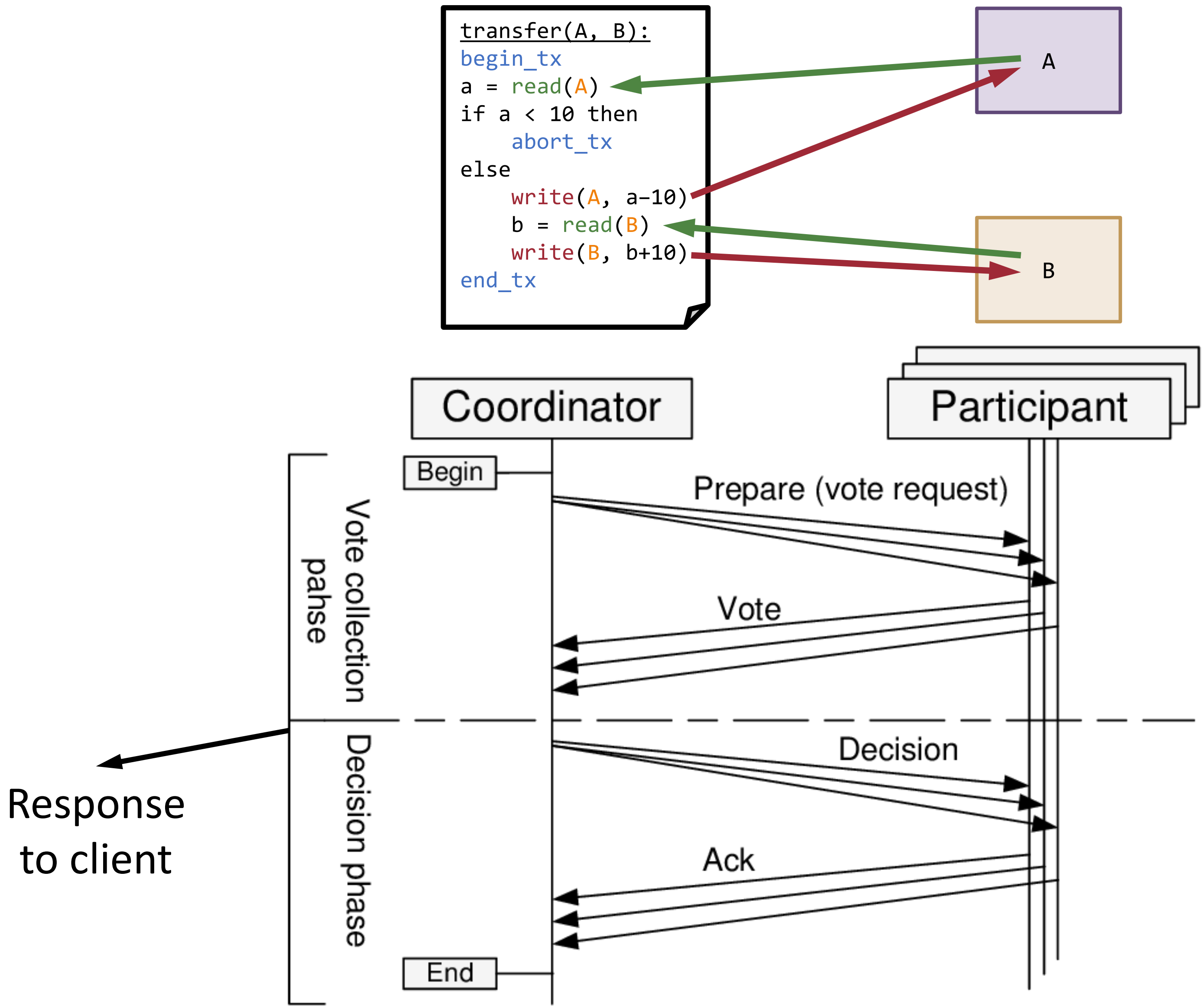
- Problems with distributed transactions
  - One participant performs all accesses but the other crashes
  - One participant performs all accesses but the other needs to abort
    - Transaction constraint fails (e.g., a < 10)
    - Cannot acquire required lock (e.g., deadlock)
    - No memory or disk 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)

Two-phase commit

- A protocol for ensuring atomic commit
- Runs after transaction execution is done



Two-phase commit protocol



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 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 can't usually 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 Commit 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

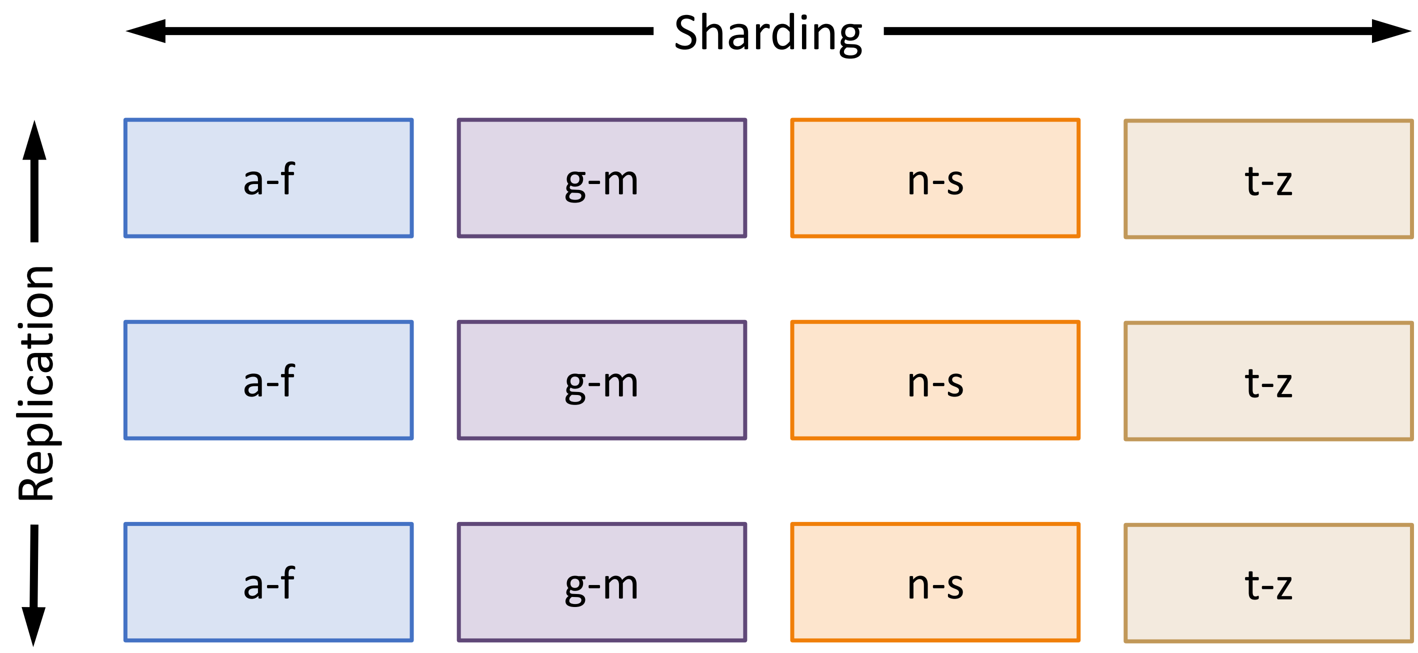
- Two-phase commit makes distributed transactions costly
  - Latency
    - Requires two additional round trips after transaction code completes
    - Votes and decision are logged to disk synchronously
  - Throughput
    - Locks are held from the time reads and writes are performed (2PC) or from prepare 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?

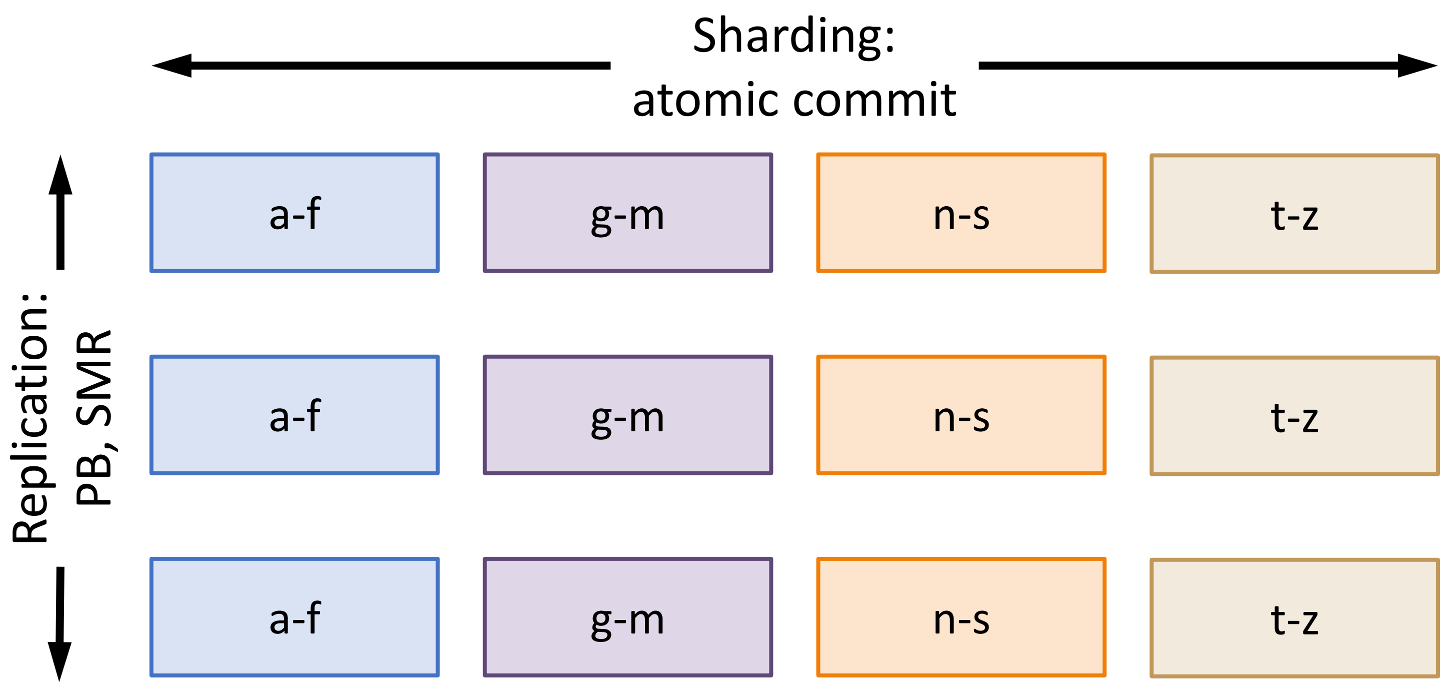


Replication, sharding, atomic commit

- 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)
  - 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)
  - All accesses execute or none (hide failures)
- Concurrency control algorithms hide concurrency
- Atomic commit protocols hide failures
  - Needed for distributed transactions
  - Require logging (at coordinator and participants)
  - Require two phases, for collecting votes, and sending decision