





Design and Engineering of Computer Systems

Lecture 38: Atomicity

Mythili Vutukuru

IIT Bombay

Atomicity: all-or-nothing

- **Atomicity**: in computer systems, some sequence of operations must appear like one atomic unit, for functional correctness.
 - Either all the operations complete, or none is executed, but no partial execution
- **Transaction**: set of operations that must execute with all-or-nothing atomicity
 - E.g., when appending to a file, several disk changes must happen together: block with new data is written, inode stores pointer to new block, free bitmap marks block as used
 - E.g., checkout from e-commerce website, billing and shipping must happen atomically
 - E.g., when transferring money from one bank account to another, debit and credit operations must happen atomically
- Failures violate atomicity: some operations in a transaction may complete, others may be left incomplete due to a component failure
- This lecture: how to ensure atomicity of transactions in spite of failures

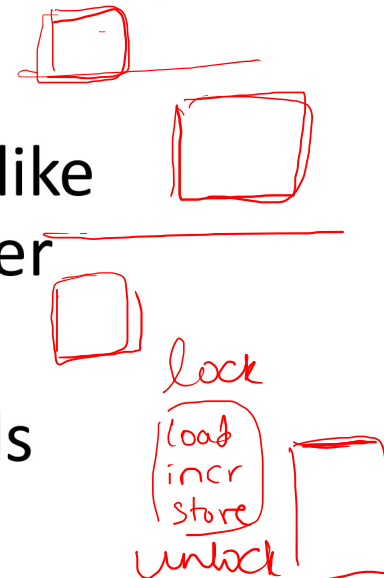
Atomicity and modularity



- Atomicity is important when building modular applications
- Example: module A sends request to module B to perform some operation, e.g., front end sends request to shopping cart server to add item to a cart
- If processing the request needs multiple operations at B, they should all be performed atomically. Why?
 - Ideally, if B completes all operations, B returns successful response to A. If request execution fails at B, no operation executed, error returned, A retries entire request
 - If B performs only some operations and not others, what response will it return? How should A retry? Makes design messy
- Design guideline: module's API should expose fine-grained services which can be executed / implemented atomically

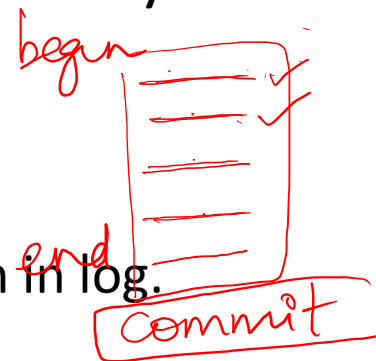
Another definition of atomicity: before-or-after

- Before-or-after atomicity: all operations in a transaction appear like one atomic unit and execute either fully before or fully after other events/transactions
- Example where before-or-after atomicity is required: updating shared counter (counter++)
 - Load counter into register, increment register, store from register to memory
 - Overlapping execution leads to incorrect result, race condition
- One way to achieve before-or-after atomicity: locking to serialize execution order, naturally prevents overlap of transactions
 - Care to be taken to avoid deadlocks with multiple locks



All-or-nothing atomicity: write-ahead logging

- **Write-ahead logging**: common technique to achieve all-or-nothing atomicity of transactions in spite of failures, using log on persistent storage
- Phases in write-ahead logging
 - Begin transaction: assign unique transaction ID, start logging
 - **Pre-commit phase**: record all operations to be performed as part of transaction in log. Original copy of data is untouched
 - **Commit point**: commit entry written to log in one atomic operation
 - **Post-commit phase**: install transaction, replay operations in log on original data
 - End transaction: clean up state, clear log entry
- How to recover from failures?
 - Failure before commit point: no changes made to original data, abort transaction
 - Failure after commit point: recover from log by replaying log entries, all operations in transaction are completed using log



begin
—
—
—

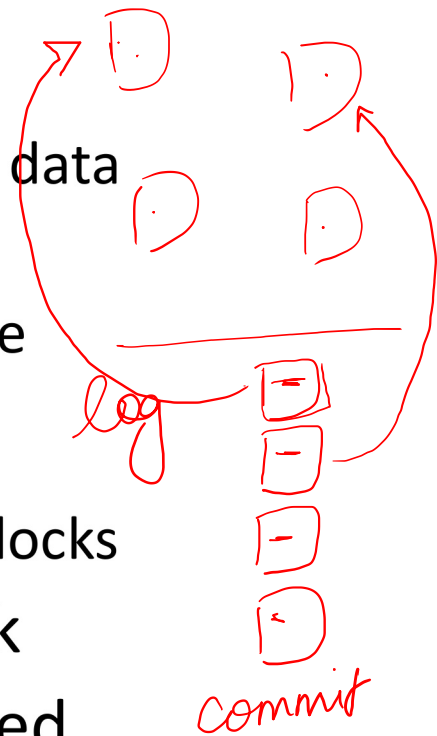
Undo logging vs. redo logging

- Technique described so far: redo logging, roll forward recovery
 - Original untouched before commit. Transaction installed only after commit point. Log used to redo changes in case of failure
- Alternate way to do write-ahead logging is undo logging
 - Pre-commit phase: log old value of data in log, directly edit original data
 - Commit point: write commit entry to log
 - Post commit phase: do nothing, clean up log entry
- How to recover from failures? (rollback recovery)
 - If failure before commit point, undo changes using old values in log, abort
 - If failure after commit point, do nothing, transaction complete
- Undoing operations may be difficult in some cases
 - Both techniques can be used depending on application

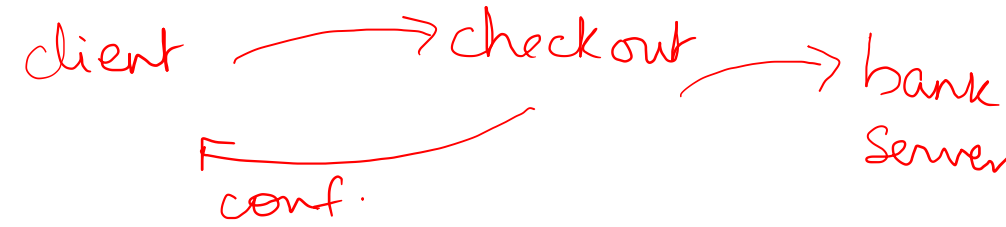


Example: crash-consistent file system

- File systems use write-ahead logging for crash consistency
 - File-related system calls update multiple disk blocks
 - All changes must happen atomically, else inconsistent file system data
 - E.g., inode pointing to data block, but data block does not have correct data
- Logging for crash consistency
 - Pre-commit phase: make changes to shadow copies of disk blocks, write changed blocks to log on disk, original blocks untouched
 - Commit point: commit entry in log
 - Post-commit phase: install transaction, make changes to original disk blocks
- Failure before commit: system call fails, no changes made to disk
- Failure after commit: system call replayed from log and completed



Example: checkout server (1)



- Consider purchase management / checkout server in e-commerce website
 - Client / frontend makes request to checkout an order via checkout server API
 - Server handles request, charges payment from user by contacting banking server, initiates shipping of order after payment completes, sends confirmation to user
- How to checkout atomically?
 - Pre-commit: create order, assign unique ID, log all order details
 - Commit point: order details successfully stored in replicated/persistent storage. Order can be confirmed to user at this point
 - Post-commit: proceed to perform operations in the order (billing, shipping)
- How to recover from failures?
 - If server fails before commit point, order not recorded in system, user will not get confirmation, user will retry later with a new order
 - If server fails after commit point, new server replica (or old server after restart) will resume execution of operations and complete checkout, user won't notice failure



Example: checkout server (2)

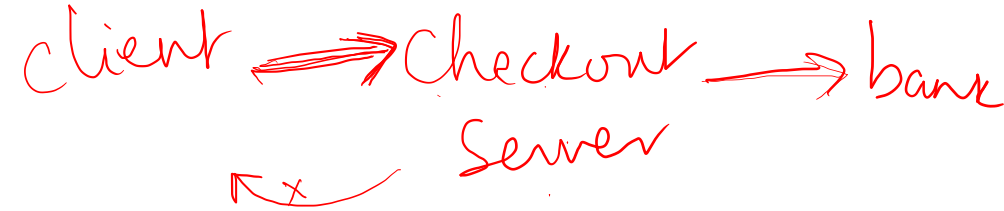


- Consider following scenario
 - Server recorded order details (commit point), starts executing order, crashes during/after billing completes but before shipping is initiated
 - New server (or restarted old server) resumes order execution, replays billing operation again at banking server, charges user twice!
- Why this complication?
 - Some operations are naturally idempotent and can be executed multiple times, e.g., replacing old disk block with new modified disk block, updating user credit card info with new details
 - Some operations should be performed exactly once, e.g., charging user's credit card
- If API is not naturally idempotent, module implementation must try to ensure idempotent semantics. How to make payment at banking server idempotent?
 - Banking server maintains a database of transaction IDs that have been billed recently
 - If billing request completed but requested again, bank returns confirmation without charging twice
- **Idempotent APIs** greatly simplify system design, allow other modules to retry/replay operations while recovering from failures, without violating correctness

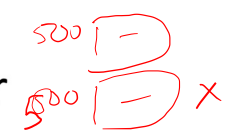
$a = 1$
 $a++$



Example: checkout server (3)



- Multi-tier apps: all tiers must ensure idempotent operations in their APIs
- Example: client → checkout server → banking server
 - Client does checkout, does not get confirmation, retries. If checkout failed previously, checkout server executes request. If checkout was completed previously but confirmation was lost, checkout server directly returns confirmation instead of executing order again
 - Checkout server performs billing at banking server, crashes in between. New checkout server retries request. Banking server charges payment only if previous payment did not complete
- Design guideline: use a unique identifier to track a request across all modules in the application, helps to avoid executing same request twice in case processing request is not idempotent operation
 - Similar to TCP sequence numbers to filter out duplicate packets at receiver
- Similar ideas used by network libraries, e.g., RPC frameworks, to guarantee exactly once RPC semantics (execute RPC exactly once even in case of failures)



Summary

- In this lecture:
 - Atomicity: all-or-nothing, before-or-after
 - All-or-nothing atomicity via write ahead logging
 - Atomicity in multi-tier applications, idempotent operations
- Think of real-life situations where atomicity and idempotent operations occur. What are some of the techniques used in such scenarios?