Distributed Transactions

Paper Notes

Before-Or-After Atomicity

- Previously, we talked about coordination via sequence coordination
  - This says that Action W must happen before Action X for correctness
  - Usually explicitly programmed using special language constructs or shared variables
    - I.e. use locks / semaphores to enforce specific ordering of operations
- Before-or-after atomicity is a more general constraint that several actions acting on the same data should not interfere with one another
  - Definition: concurrent actions have the before-or-after property if their effect from the invokers is the same as if the actions occurred either completely before or completely after one another
  - The programmer of an action with the before-or-after property does not necessarily know all other actions that might use shared variables
- Another way of saying this is that the results must be serializable
- The important distinction is that sequence coordination says two actions must be entirely separate while before-or-after says that they can occur at the same time and have interleaved instructions, but they must have the same state at the end
- In some applications, we may want stronger correctness requirements:
  - External time consistency means that if we have receipts that one action happened before another, then the serialization order must obey this (something like linearizability)
  - Sequential consistency means that when a processor performs multiple instructions from the same source, the result should be the same as if the instructions were in the original order

Simple Locking

- A locking discipline
- Two rules:
  - Each transaction must acquire a lock for every shared data object it intends to read / write
  - It may release its locks only after the transaction installs its last update
- A transaction has a lock point, which is the first instant at which it has acquired all of its locks
  - The collection of locks it has acquired is its lock set
- A lock manager can be used to maintain the current lockset and intercept variable accesses to verify that they are in the current lockset
  - Also intercepts calls to commit / abort so it can then release all locks
- Can miss some opportunities for concurrency by unnecessary locking
  - Two operations that both want to read cannot be made concurrent
  - Also we lock every variable that we **might** need to read, so we could get extra performance by only locking what we will actually read

Two Phase Locking

- Another locking discipline avoids the requirement that a transaction must know in advance which locks to acquire
- Widely used, but harder to argue it is correct
- A transaction acquires locks as it proceeds, and it may read / write a data object as soon as it acquired a lock
  - A transaction may not release any locks until it passes its lock point (acquired all of its locks)
  - Number of locks acquired monotonically increases and then decreases (which is where two phases comes from)

- Orders concurrent transactions so they produce results as if they had been serialized in the order in which they reached their lock points
- A lock manager can implement by intercepting all calls to read / write and acquiring a lock on the first use of each shared variable
  - Holds the locks until the commit / abort occurs
- Potentially allows for more concurrency than simple locking, but there are still cases where it prevents some concurrency

Multiple-Site Atomicity / Distributed Two-Phase Commit

- We previously talked about atomicity / concurrency where we have several processes running on the same system
- Now we look at executing component transactions at several sites separated by a best-effort network (i.e. the Internet)
  - Atomicity is difficult because messages used to coordinate transactions can be lost / delayed / duplicated
  - We could use RPCs to execute target code but it doesn't by itself guarantee atomicity
    - Even if we use persistent senders (at-least-once execution) and duplicate suppression (at-most-once execution), this isn't guaranteed to work as shown below
  - Instead we use a variant of two-phase commit combined with persistent senders, duplicate suppression, and single-site transactions
  - We assume that each site is capable of local transactions that satisfy all-or-nothing atomicity and before-or-after atomicity
- Multiple-site atomicity is correct iff all sites do the same thing (either they all commit or they all abort)

Description

- Suppose A wants to send jobs X, Y, Z to be performed by B, C, and D
  - Either all of these are done or none of them are done
  - Can't just use RPCs raw because some could perform the action while others don't
- Procedure:
  - A first sends each B, C, D the instruction to do X, Y, Z respectively
  - B, C, D respond saying that they are ready to commit
  - When A receives all responses, they respond to everyone to tell them to prepare to commit
  - B, C, D then tentatively commit and ask A if they have decided to commit
    - If they don't receive a response, then they continue asking A
    - They cannot do anything on their own and instead must wait for a response from A
    - They go into a "Prepared" state
  - A then responds that they have committed
    - They must remember that they have committed forever because if B, C, D ask again if they have committed, then A has to respond again
  - B, C, D then commit and then can switch out of Prepared state
- Resilient against temporary failures
- Some versions have a fourth acknowledgment record so that A can know when to throw out its outcome record
- After A commits, then they know that B, C, D will eventually get their result when they reboot / recover from failure
- This system has the coordinator as a single point of failure
  - The coordinator has to decide when to abort the operation
- This system cannot coordinate actions to happen at the same time (Two General's Paradox)

Lecture Notes

- So far, we've covered distribution for fault tolerance
  - Many servers trying to look like a single one

- Now we want to use many servers for performance
  - We can split the data (shard) over multiple servers for parallelism
  - Fine as long as clients use data items one at a time
  - But what if an operation involves records in different shards?
    - Need to perform a distributed transaction
- Correct behavior for a transaction: ACID
  - Atomic (either writes entirely or doesn't write at all)
  - Consistent
  - Isolated (No interference between actions (serializable))
  - Durable (Committed writes are permanent)
- Once we have ACID transactions, it's like its magic
  - Programmer writes serial code and system automatically adds correct locking / fault tolerance
  - We do get a performance hit from transactions and some don't
  - SQL databases provide transactions
- A transaction can abort if something goes wrong
  - Result should be as if the transaction never executed
  - Application can decide whether they want to retry

Distributed Transactions

- Two components:
  - Concurrency Control - isolated, serializable execution of concurrent transactions on a single DB
    - Two types: pessimistic and optimistic
      - Pessimistic: locks records before use, so any concurrent accesses cause delays
      - Optimistic: use records without locking, when we commit, we check if the reads / writes were serializable
        - If we learn that our reads / writes were bad, then we abort and retry
    - Pessimistic is faster if conflicts are frequent, optimistic is faster if rare
    - We'll look at optimistic concurrency control another time (FaRM)
    - Two-phase locking above is one way to implement pessimistic concurrency control
      - See above
      - Some extra details:
        - Each database record has a lock
        - DB may automatically abort to cure deadlock (has to automatically detect these i.e. through cycles or timeout)
  - Atomic Commit
    - Two-phase commit works by having a transaction coordinator (TC) managing the operation
      - Bad reputation:
        - Multiple rounds of messages
        - Slow disk writes
        - Locks are held between messages, blocking actions
        - TC crashes can cause indefinite blocking with locks held
      - Usually only used in a single small domain
- 2PC is for ensuring every participant has to do something, but that something is different among all of them
  - On the other hand, Raft is used to get high availability, so it solves a different problem
  - Each of the servers can run Raft to combine both distributed commits and high availability