1. Distributed Transactions
    1. Background
    2. Concurrency Control
    3. Consolidate with Raft
    4. Reference

Distributed Transactions

Background

Problem:

- lots of data records, sharded on multiple servers, lots of clients

Correct behavior of a xactions: ACID

- A: Atomicity, all writes or none, despite failures
- C: obeys application-specific invariants
- I: Isolation, no interference between xactions -- serializable
- D: Durability, committed writes are permanent

How to test whether the xactions is serializability:

- example transactions

```
x and y are bank balances -- records in database tables
x and y are on different servers (maybe at different banks)
x and y start out as $10
T1 and T2 are transactions
  T1: transfer $1 from x to y
  T2: audit, to check that no money is lost
T1:             T2:
begin_xaction   begin_xaction
  add(x, 1)       tmp1 = get(x)
  add(y, -1)      tmp2 = get(y)
end_xaction       print tmp1, tmp2
                end_xaction
```

- execute concurrent transactions T1 and T2

```
T1; T2 : x=11 y=9 "11,9"
T2; T1 : x=11 y=9 "10,10"
the results for the two differ; either is OK
```

Concurrency Control

Distributed transactions have two big components:

- concurrency control (to provide isolation/serializability)
- atomic commit (to provide atomicity despite failure)

Two classes of concurrency control for transactions:

- pessimistic:
  - lock records before use
  - conflicts cause delays (waiting for locks)
- optimistic:
  - use records without locking
  - commit checks if reads/writes were serializable
  - conflict causes abort+retry
  - called Optimistic Concurrency Control (OCC)
- pessimistic is faster if conflicts are frequent
- optimistic is faster if conflicts are rare

"Two-phase locking" is pessimistic:

- a transaction must acquire a record's lock before using it
- a transaction must hold its locks until *after* commit or abort

Process:

```
TC              A            B
|----put------>|(lock data)
|-----------get---------->|(lock data)
|
|
|---prepare--->|
|----------prepare-------->|
|
|<---yes/no----|
|<-----------yes/no--------|
|
|
|-commit/abort->|(commit/abort and release lock)
```

```
  |---------commit/abort---->|(commit/abort and release lock)
  |<----ack-----|
  |<-------------ack---------|
```

Failure tolerance:

- What if B crashes and restarts?
    - If B sent YES before crash, B must remember (despite crash)! Because A might have received a COMMIT and committed. So B must be able to commit (or not) even after a reboot.
        - write log to disk
- What if TC crashes and restarts?
    - If TC might have sent COMMIT before crash, TC must remember! Since one worker may already have committed.
        - write log to disk before sending COMMIT msgs.
        - repeat COMMIT if it crashes and reboots, or if a participants asks.
- What if TC never gets a YES/NO from B?
    - Perhaps B crashed and didn't recover; perhaps network is broken.
        - TC can time out, and abort (since has not sent any COMMIT msgs).
- What if B times out or crashes while waiting for PREPARE from TC?
    - B has not yet responded to PREPARE, so TC can't have decided commit
        - B can unilaterally abort, and release locks
        - respond NO to future PREPARE
- What if B replied YES to PREPARE, but doesn't receive COMMIT or ABORT?
    - B cannot decide to abort it, because TC might have gotten YES from both, and sent out COMMIT to A, but crashed before sending to B.
        - cannot do anything, just wait for TC came back

Two-phase commit perspective:

- Used in sharded DBs when a transaction uses data on multiple shards
- slow: multiple rounds of messages
- slow: disk writes
- locks are held over the prepare/commit exchanges; blocks other xactions
- TC crash can cause indefinite blocking, with locks held

Consolidate with Raft

Difference between Raft:

- Use Raft to get high availability by replicating, Raft does not ensure that all servers do something. Only a majority have to be alive.
- Use 2PC when each participant does something different, 2PC does not help availability

Consolidate with Raft, achieve both high availability *and* atomic commit:

- The TC and servers should each be replicated with Raft
- It is the basic theory of Spanner

Reference

- Principles of Computer System Design, Chapter 9
- mit-course-note: distributed transactions