# 6.824 2006 Lecture 13: Two-Phase Commit

General topic: coordinating servers in a distributed system.
- Prev lectures: agree on order of updates to replicas, no failures.
- Today: general-purpose agreement on some action, w/ failures.
- Next few lectures: agree on updates to replcias, w/ failures.

Example:
- Transfer money from bank A to bank B.
  - Debit at A, credit at B, tell client "ok".
- We want both to do it, or both not to do it.
- We *never* want only one to act.
  - We'd rather have nothing happen (this is important).
- We want an "Atomic Commit Protocol"

Bigger transaction processing picture: we want two kinds of atomicity:
- Serializability, as if in some order, locking
- Recoverability, all or nothing, no partial results
- Today is really about recoverability
  - I'll assume some external force serializes
  - Perhaps a lock server forcing one-at-a-time transactions
  - Perhaps there's only one source of transactions

Atomic Commit is hard
- "I'll commit if you commit"
- What if I don't hear back from you?
- Neither party can finally decide

Straw Man:
- invent a Transaction Coordinator as single authoritative entity
- four entities: client, Transaction Coordinator, Bank A, Bank B
- client sends "go" to TC
- TC sends "debit" to A
- TC sends "credit" to B
- TC reports "ok" to client.
- Timeline... No ACKs.

How can this go wrong?
- Maybe there's not enough money in A's bank account.
- Maybe B's bank account no longer exists.
- Maybe the network link to B is broken.
- Maybe A or B has crashed.
- Maybe TC crashes between sending the messages.

We want two properties:
- TC, A, and B each have a notion of committing
- Correctness:
  - if one commits, no one aborts.
  - if one aborts, no one commits.
- Performance:
  - if no failures, and A and B can commit, then commit.
  - if failures, come to some conclusion ASAP.

Let's do correctness first.

Correct atomic commit protocol:
- TC sends "prepare" messages to A and B.

```
  A and B respond, saying whether they're willing to commit.
  If both say "yes", TC sends "commit" messages.
  If either says "no", TC sends "abort" messages.
  A/B "decide to commit" if they get a commit message.
    I.e. they actually change the bank account.

Why is this now correct?
  Neither can commit unless they both agreed.

What about performance?
  Crashes or message losses can still prevent completion.

We have two types of problems:
  Timeout. I'm up, but I don't recv a msg I expect.
    Maybe the other host crashed. Maybe the network is broken.
    We cannot usually tell the difference, so must be correct in either
case.
  Reboot. I crashed, and I'm rebooting, and I need to clean up.

Let's deal with timeout first.
  Where do hosts wait for messages?
  1) TC waits for yes/no.
  2) A and B wait for commit/abort.

Handling TC timeout waiting for yes/no.
  TC has not sent any "commit" messages.
  So TC can safely abort, and send "abort" messages.
  We've preserved correctness, but sacrificed performance.
  Because (maybe) A and B were both prepared to commit,
    but we lost one of the messages,
    we could have committed, but TC didn't know it.
  TC is being conservative.

Handling A/B timeout while waiting for commit/abort.
  Let's talk about just B (A is symmetric).
  If B voted "no", it can unilaterally abort.
  So what if B voted "yes"?
  Can B unilaterally decide to abort?
    No! TC might have gotten "yes" from both,
    and sent out "commit" to A, but crashed before sending to B.
    So then A would commit and B would abort: incorrect.
  B can't unilaterally commit, either:
    A might have voted "no".

B could just wait forever until it gets commit/abort from TC.
  But you can do better than that:

Termination protocol for B if it voted "yes":
  B sends "status" request message to A.
    Asks if A knows whether transaction should commit.
  If B doesn't hear reply from A: no decision, wait for TC.
  If A received "commit" or "abort" from TC: B decides the same way.
    Can't disagree with TC...
  If A hasn't voted yes/no yet: B and A both abort.
    TC can't have decided "commit", so it will eventually hear from A
or B.
  If A voted "no": B and A both abort.
```

```
    TC can't have decided "commit".
  If A voted "yes": no decision possible!
    TC might have decided "commit" and replied to client.
    Or TC might have timed out and aborted.
    A and B must wait for the TC.

What have we achieved?
  Can resolve some timeout situations w/ guaranteed correctness.
  But sometimes A and B must block.
  Due to TC failure, or failure of TC's network connection.

How to handle crash/reboot?
  Cannot back out of a commit if already decided.
  TC crashes just after deciding "commit".
  A/B crash just after sending "yes".
  Big trouble if they reboot and don't remember saying "yes"!
    They might change their minds after the reboot.
    Or *even after everybody re-starts* they may not be able to decide!
  If all nodes know state before crash, you can use termination
protocol.
    But also talk to TC, which may know that it committed.

How do you know what state you were in before the crash?
  Assume non-volatile memory, perhaps a disk.
  But do you write disk, then send "yes" message? Or "commit" if TC?
    Or send message and then update the state on the disk?
  We cannot send the message before writing the disk:
    Since then we might change our mind after the reboot.
    I.e. B might have voted "yes", then reboot, then "no".
  Does it work to write the disk before sending the message?
    For TC w.r.t. "commit", and A/B w.r.t. "yes".

Recovery protocol w/ non-volatile state:
  If you're the TC, and there's no "commit" on disk, abort.
    Because no commit on disk -> you didn't send any "commit" messages.
  If you're A/B, and no "yes" on disk, abort.
    No "yes" on disk -> didn't vote yes -> nobody could have committed.
  If you're A/B, and there is a "yes" on your disk:
    Ordinary termination protocol, might block.
  If everyone has rebooted and is reachable, you can decide.
    Based just on whether TC has "commit" on its disk.

This protocol is called "two-phase commit".
  What properties does it have?
  1. All hosts that decide reach the same decision.
  2. No commit unless everyone says "yes".
  3. No failures and all "yes", then commit.
  4. If failures, then repair, wait long enough, then some decision.
```