Two-Phase Commit

Brad Karp
UCL Computer Science

CS GZ03 / M030
27th October, 2010

Context: Sharing and Failures

- Thus far:
  - NFS: share one filesystem among many clients, with explicit communication, caching, and (weak) consistency
  - Ivy: share memory among many CPUs, with implicit communication, read-only sharing, and stronger consistency
- **What happens when components in distributed system fail?**

Challenge: Agreement in Presence of Failures

- Two servers must each take an action in distributed system
- Can we ensure they agree to do so?
- Example: transfer money from bank A to bank B
  - Debit A, credit B, tell client "OK"
- Want both to do it or neither to do it
- Never want only one side to act
  - **Better if nothing happens!**
- Goal: Atomic Commit Protocol

Transaction Processing Context: Two Kinds of Atomicity

- Serializability:
  - Series of operations requested by users
  - Outside observer sees them each complete atomically in some complete order
  - Requires support for locking
- Recoverability:
  - Each operation executes completely or not at all; "all-or-nothing semantics"
  - No partial results

Transaction Processing Context: Two Kinds of Atomicity

Today's topic: **recoverability**

Assume for now some external entity serializes:

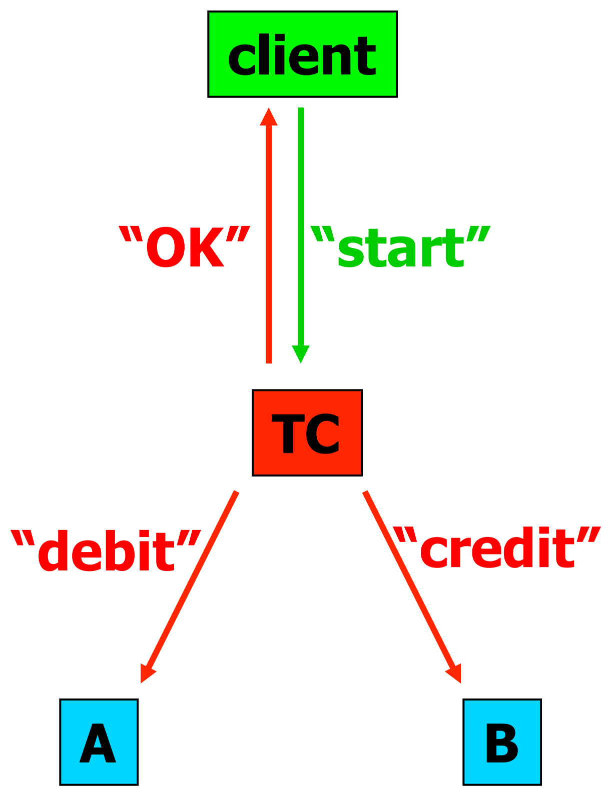
- Lock server may force transactions to execute one at a time
- Or maybe only one source of transactions

- Recoverability:
  - Each operation executes completely or not at all; "all-or-nothing semantics"
  - No partial results

Atomic Commit Is Hard!

- A -> B: "I'll commit if you commit"
- A hears no reply from B
- Now what?
- **Neither party can make final decision!**

Straw Man Atomic Commit Protocol



- Create Transaction Coordinator (TC), single authoritative entity
- Four entities: client, TC, Bank A, Bank B
- Client sends "start" to TC
- TC sends "debit" to A
- TC sends "credit" to B
- TC reports "OK" to client

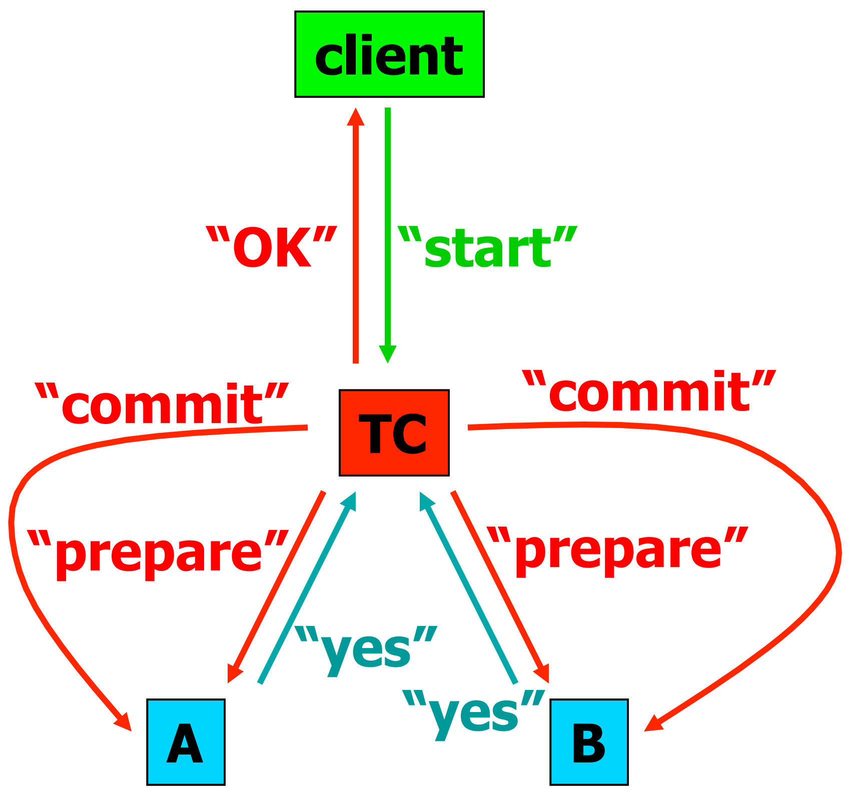
Failure Scenarios

- Not enough money in A’s bank account
  - A doesn’t commit, B does
- B’s bank account no longer exists
  - A commits, B doesn’t
- Network link to B broken
  - A commits, B doesn’t
- One of A or B has crashed
  - Other of A or B commits, A or B doesn’t
- TC crashes between sending to A and B
  - A commits, B doesn’t

Atomic Commit: Defining Desirable Properties

- TC, A, and B have separate notions of committing
- Safety
  - (Really, “correct execution”)
  - If one commits, no one aborts
  - If one aborts, no one commits
- Liveness:
  - (In a sense, “performance”)
  - If no failures, and A and B can commit, then commit
  - If failures, come to some conclusion ASAP

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 and B "decide to commit" if they receive a commit message.
  - In example, "commit" means "change bank account"

Protocol's Safety, Liveness?

- **Why is previous protocol correct (i.e., safe)?**
  - Knowledge centralized at TC about willingness of A and B to commit
  - TC enforces both must agree for either to commit
- **Does previous protocol always complete (i.e., does it exhibit liveness)?**
  - No! What if nodes crash or messages lost?

Liveness Problems

- Timeout
  - Host is up, but doesn’t receive message it expects
  - Maybe other host crashed, maybe network dropped message, maybe network down
  - Usually can’t distinguish these cases, so solution must be correct in all!
- Reboot
  - Host crashes, reboots, and must “clean up”
  - i.e., want to wind up in correct state despite reboot

Fixing Timeouts (1)

- **Where in protocol do hosts wait for messages?**
  - TC waits for "yes"/"no" from A and B
  - A and B wait for "commit"/"abort" from TC
- Making progress when TC waits for "yes"/"no"
  - TC not yet sent any "commit" messages
  - TC can safely abort, send "abort" messages
  - Preserved safety, sacrificed liveness (how?)
  - Perhaps both A, B prepared to commit, but a "yes" message was lost
  - Could have committed, but TC unaware!
  - Thus, TC is conservative

Timeouts (2): Progress when A or B Times Out Awaiting "commit"/"abort"

- wlog, consider B (A case symmetric)
- If B voted "no", can unilaterally abort; TC will never send "commit" in this case
- **What if B voted "yes"? Can B unilaterally abort?**
  - No! e.g., TC might have received "yes" from both, sent "commit" to A, then crashed before sending "commit" to B
  - Result: A would commit, B would abort; incorrect (unsafe)!
- **Can B unilaterally commit?**
  - No! A might have voted "no"

Timeouts (3): Progress when A or B Times Out Awaiting “commit”/”abort”

- Blocking “solution”: B waits forever for commit/abort from TC
- Better plan: termination protocol for B if voted “yes”

Timeouts (4): Termination Protocol When B Voted "yes"

- B sends "status" request message to A, asking if A knows whether transaction should commit
- If no reply from A, **no decision;** wait for TC
- If A received "commit" or "abort" from TC, B decides 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"; 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
  - TC might have timed out and aborted
  - A and B must wait for TC

Timeout Termination Protocol Behavior

- Some timeouts can be resolved with guaranteed correctness (safety)
- Sometimes, though, A and B must block
  - When TC fails, or TC’s network connection fails
  - Remember: TC is entity with centralized knowledge of A’s and B’s state

Problem: Crash-and-Reboot

- Cannot back out of commit once decided
- Suppose TC crashes just after deciding and sending “commit”
  - What if “commit” message to A or B lost?
- Suppose A and/or B crash just after sending “yes”
  - What if “yes” message to TC lost?
- If A or B reboots, doesn’t remember saying “yes”, big trouble!
  - Might change mind after reboot
  - Even after everyone reboots, may not be able to decide!

Crash-and-Reboot Solution: Persistent State

- If all nodes know their pre-crash state, can use previously described termination protocol
- A and B can also ask TC, which may know it committed
- Preserving state across crashes:
  - Need non-volatile memory, e.g., a disk
  - What order:
    - **write disk, then send "yes" message if A/B, or "commit" if TC?**
    - **or vice-versa?**

Persistent State across Reboots (2)

- Cannot send message before writing disk
  - Might then reboot between sending and writing, and change mind after reboot
  - e.g,. B might send "yes", then reboot, then decide "no"
- **Can we write disk before sending message?**
  - For TC, write "commit" to disk before sending
  - For A/B, write "yes" to disk before sending

Revised Recovery Protocol using Non-Volatile State

- TC: after reboot, if no "commit" on disk, abort
  - No "commit" on disk means you didn't send any "commit" messages; safe
- A/B: after reboot, if no "yes" on disk, abort
  - No "yes" on disk means you didn't send any "yes" messages, so no one could have committed; safe
- A/B: after reboot, if "yes" on disk, use ordinary termination protocol
  - Might block!
- If everyone rebooted and reachable, can still decide!
  - Just look at whether TC has "commit" on disk

Two-Phase Commit Protocol: Summary of Properties

- "Prepare" and "commit" phases: Two-Phase Commit (2PC)
- Properties:
  - Safety: all hosts that decide reach same decision
  - Safety: no commit unless everyone says "yes"
  - Liveness: if no failures and all say "yes," then commit
  - Liveness: if failures, then repair, wait long enough, eventually some decision

Two-Phase Commit Protocol: Summary of Properties

- “Prepare” and “commit” phases: Two-Phase Commit (2PC)
- Properties:

Theorem [Fischer, Lynch, Paterson, 1985]: no distributed asynchronous protocol can correctly agree (provide both safety and liveness) in presence of crash-failures (i.e., if failures not repaired)

  – Liveness: if failures, then repair, wait long enough, eventually some decision