CSE 451: Operating Systems
Spring 2009

Lecture 17
Two-phase commit

Steve Gribble

A fundamental problem

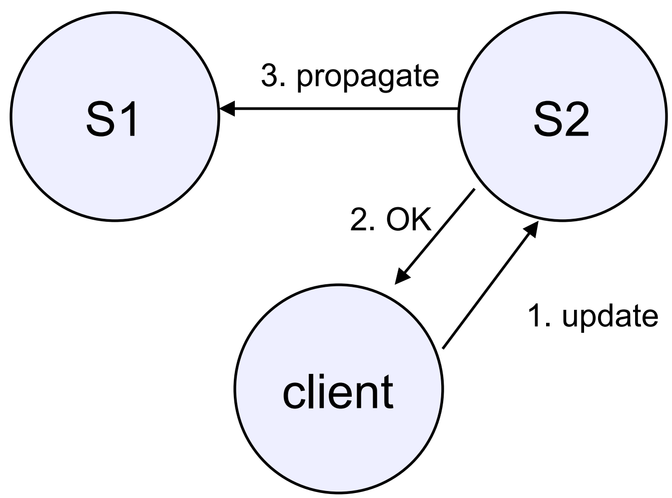
- Consider a client/server architecture
  - what happens to the service if a server crashes?
    - software failure, OS failure, hardware failure, power outage, earthquake, …
- Replication to the rescue
  - key idea: instead of having one server providing service to clients, have multiple servers providing the same service
    - each of the servers are called replicas
    - given N replicas, if one crashes, N-1 can still provide service
      - this assumes independent failures
  - replication therefore improves availability
    - however, it introduces a new problem: keeping replicas consistent with each other in the face of updates

Some quick math for the curious

- assume N replicas
  - assume a specified mean time between failure (MTBF)
    - with exponentially distributed failure arrivals
    - (in other words, a completely random process)
  - assume a specified mean time to repair (MTTR)
- what is the reliability of the overall system?
  - $MTBF_{system} \quad \alpha \quad \frac{MTBF_{replica}^{N}}{MTTR_{replica}}$
  - note that repair is a crucial part of the system!

The Replica Consistency problem

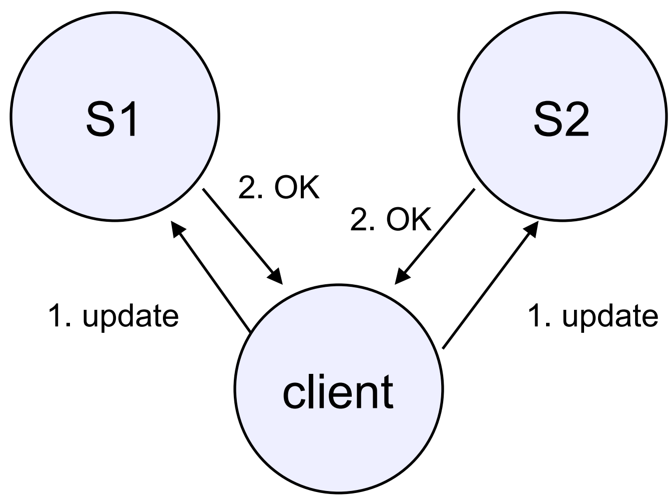
- Imagine we have two “bank” servers, and a client that updates its bank account
  - naïve replication strategy: client updates a random server. After update, the randomly chosen server propagates change to other server.
    - master/slave replication



- what are all the things that can go wrong?

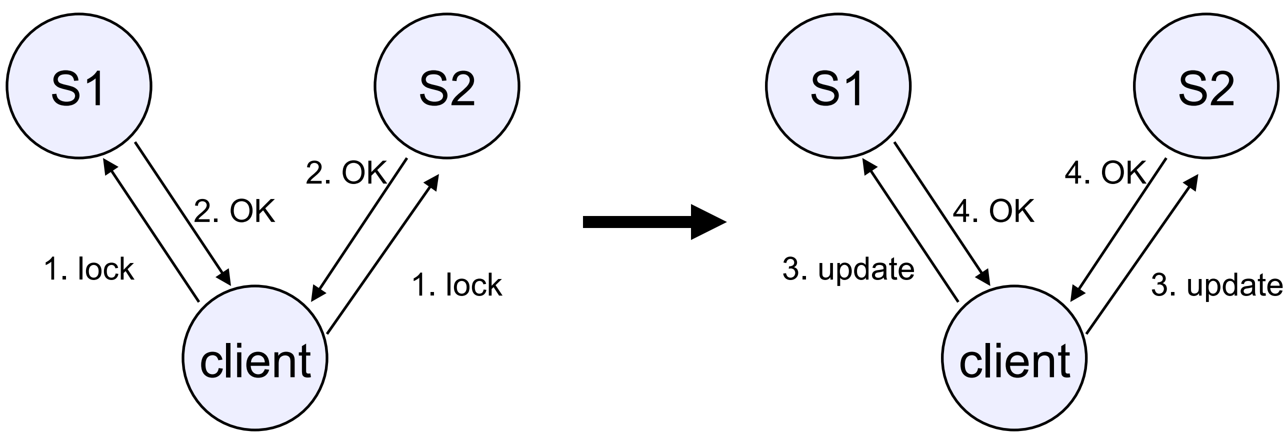
What are we to do?

- One (of many) problems is that servers can have different views of the data at the same time
  - this is the very definition of inconsistency!
  - even worse, simultaneous updates can stomp on each other
    - inconsistency is never resolved
- Idea: update both servers at once?



But there are races…

- Two clients issuing updates at same time
  - messages may arrive in different orders at different servers
    - e.g. message #1 = “turn on light”, message #2 = “turn off light”
    - what’s the state of the light switch at each server?
- How did we deal with races in multithreaded code?
  - critical sections, mutual exclusion via locks:



More problems…

- But what about:
  - network failure, or network delays
  - client failure
  - server failure
  - deadlock

Consensus

- Updating replicas is an example of a more general problem --- ***consensus in a distributed system***
  - conditions under which consensus is possible depends on assumptions and requirements
  - assumptions:
    - network: synchronous, asynchronous, or partially synchronous?
    - participants: failure-free, fail-stop, or byzantine?
  - requirements:
    - can you tolerate temporary periods of inconsistency?
    - should the system be ***wait-free,*** or is it OK for some processes to block waiting for some other process (or the network) to recover?

Bad news, good news

- The bad news: the real world is messy
  - networks are asynchronous
    - wait-free consensus **provably impossible** in an asynchronous network, even if you assume fail-stop failures, and even if you assume at most a single failure!
  - failures are byzantine, not fail-stop
    - must assume adversarial behavior
- The good news: we can cope
  - OK, networks are really partially synchronous (timing bounds exist in practice)
  - OK, can assume fail-stop in some scenarios (e.g., within a Google data center)
  - OK, can handle byzantine failures with some cost and engineering

Two-phase commit

- Goal: update all replicas atomically
  - either everybody commits update, or everybody aborts
  - no inconsistencies (including races from multiple clients)
  - even in the face of network and host failures
- Assumptions
  - synchronous network
  - assume no byzantine failures (fail-stop)
  - willing to wait (block until recovery) in some cases
- What do we get?
  - "**weak termination**:" if there are no failures, then all processes eventually decide
  - but not "**strong termination**:" all non-faulty processes eventually decide (need three-phase commit for this)

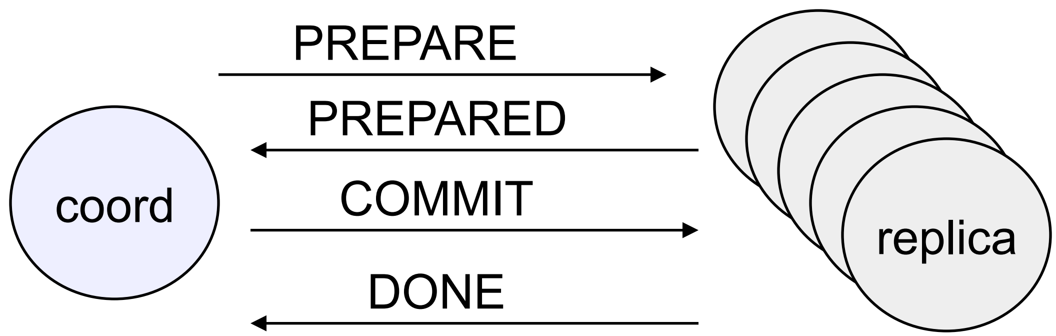
Terminology

- **coordinator**
  - software entity that shepherds process
  - client in our example, not necessarily always so
- **replica**
  - software entity to be updated by coordinator
  - coordinator can be a replica as well, if you like
- **ready to commit**
  - side-effects of update are safely stored on durable, secondary storage
  - if a replica is ready to commit, then even if it crashes, it can continue with two-phase commit after it recovers

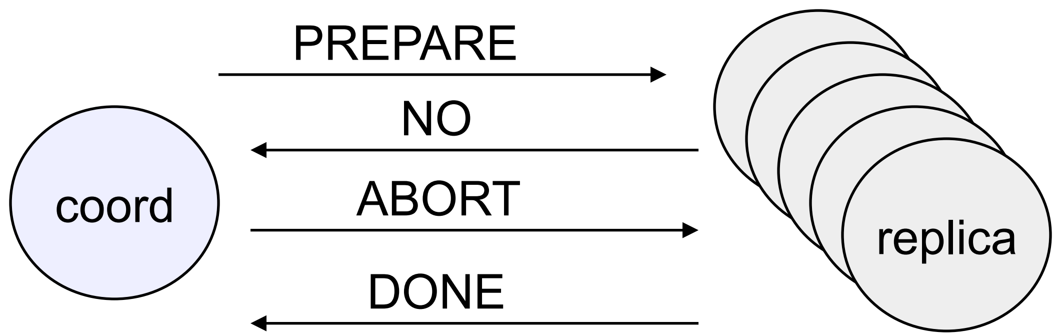
The Protocol

- Phase 1:
  - coordinator sends a PREPARE message to each replica
  - coordinator waits for all participants to vote
  - each participant:
    - votes PREPARED if it is ready to commit
      - also locks data item(s) being updated
    - votes NO for any reason
      - including inability to grab a lock
    - may delay voting arbitrarily…
- Phase 2:
  - if coordinator receives PREPARED from all replicas, it decides to commit. if not, it decides to abort.
    - at this point, the "transaction" or update is over
  - coordinator sends its decision to all participants
    - COMMIT or ABORT
      - participant marks decision, releases lock
  - participants acknowledge receipt with DONE

Outcome #1: COMMIT



Outcome #2: ABORT

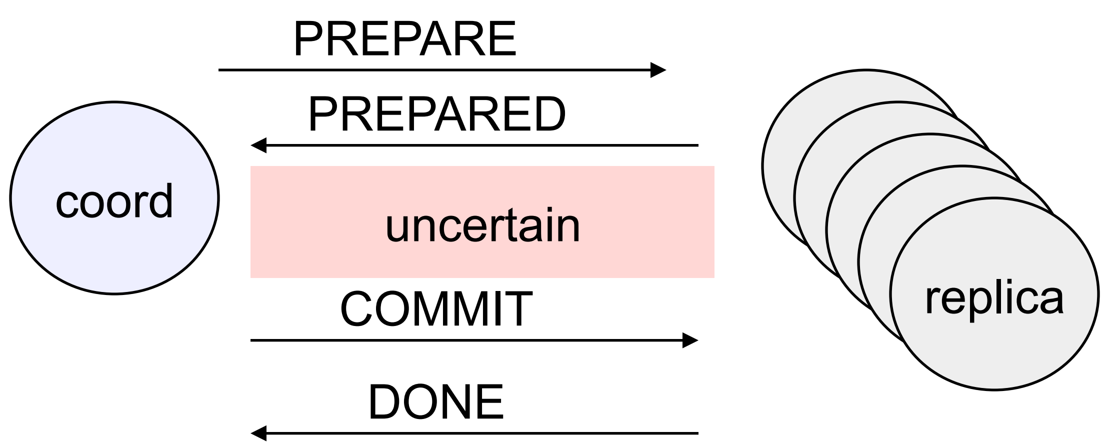


Performance

- In the absence of failures, 2PC makes a total of 1.5 round-trips of messages before decision is made
  - prepare
  - vote to prepare
  - commit/abort
  - (note that the “DONE” is just for bookkeeping, it doesn’t affect response time)

Uncertainty

- Before it votes, a replica can unilateraily abort
- After it votes PREPARED and before it receives the coordinator's decision, a replica is in an uncertain condition.
  - it can't either commit or abort until it hears from coordinator



More uncertainty

- Note that the coordinator is never uncertain
  - it can always unilaterally abort, until it sends out a COMMIT
- If a participant fails or is partitioned during uncertain period…
  - it must contact coordinator to discover decision after recovery or network repair
    - implies coordinator must keep track of decisions
    - for how long?

Failure handling

- Failure is detected with timeouts
  - must eventually rely on timeouts in a distributed system
- If participant times out waiting for PREPARE
  - it can simply abort
- If coordinator times out waiting for a vote
  - it can simply abort
- If participant times out waiting for a decision
  - it becomes “blocked”
    - punt to some other resolution protocol
    - simplest one: wait for coordinator to recover
- If coordinator times out waiting for a done
  - ?