Distributed Systems

Lec 16: Agreement in Distributed Systems: Two-phase Commit Problems, Three-phase Commit

Slide acks: Jinyang Li, “The Paper Trail”
Agreement in Distributed Systems

• The crown problem of distributed systems
  – A.k.a. consensus

• Despite having different views of the world, all nodes in a distributed system must act in concert, e.g.:
  – All replicas that store the same object O must apply all updates to O in the same order (consistency)
  – All nodes involved in a transaction must either commit or abort their portion of the transaction (atomicity)

• All that, despite FAILURES
  – Nodes can restart, die, be slow
  – Networks can be slow, as well (but we assume they’re reliable here, i.e., all network messages are eventually received)
The Agreement Problem

- Some nodes propose values (or actions) by sending them to the others
- All nodes must decide whether to accept or reject those values

Examples of values to agree on:
- Whether or not to commit a transaction to a DB
- The value of the clock
- The leader that will coordinate some higher-level protocol
- Who has a lock in a distributed lock service among multiple clients that request it almost simultaneously
- Whether to move to the next stage of a distributed alg. (a barrier)
Agreement Requirements

- **Safety** (correctness)
  - All nodes agree on the same value
  - The agreed value X has been proposed by some node

- **Liveness** (fault tolerance, availability)
  - If less than some fraction of nodes crash, the rest should still reach agreement

- I.e., agreement aims to give the behavior of a single machine with the fault-tolerance of multiple machines
Failure Models

• For these classes, we define agreement in the context of two failure models:
  
  • **Synchronous** systems: machines and networks can only be delayed by a bounded time
    — I.e., using a sufficiently large timeout, you can tell with certainty whether the machine crashed or it or the network is just slow
  
  • **Asynchronous** systems: machines and networks can be arbitrarily delayed ← more general
    — There’s no way you can tell whether a machine has crashed or is just slow

• We’ll see that different safety/liveness properties are possible under different models
What We’ve Learned So Far

• We’ve already been discussing about agreement, e.g.:
  – Logical clocks are a form of agreement (what’s the time?)
  – Distributed mutex algos (who has lock?)
  – Two-phase commit (commit or abort?)

• However, none of the algorithms thus far are particularly fault-tolerant (or live during failures)
  – Distributed mutex algo block when any node crashes
  – Two-phase commit (2PC) blocks when TC crashes (we’ll see example today)

• Last time, we talked about fault recovery
  – Recovering 2PC (will finish today)
Today

• Fault recovery is important, but is insufficient, because recovery can be very slow
  – E.g., the 2PC coordinator may be down for a long time before it reboots, you don’t want the whole protocol to wait for it

• You want fault tolerance
  – I.e., high availability despite concurrent faults
  – (The ability to recover from faults is still important, so that a failed replica can re-join the group after reboot as seamlessly as possible)

• Today’s (and next time’s) plan:
  – Finish discussion about recovery-enabled 2PC
  – Talk about the fault-tolerance limitations of 2PC
  – Introduce 3 phase commit (3PC)
  – Introduce Paxos
Recovery-enabled
Two-Phase Commit

(repeat from last time’s slides, as we left them uncovered)
2PC (with consensus terminology)

Phase 1: proposal

- proposal (e.g., “commit transaction tid”)
- vote (yes/no)

TC → A → B → C → D

Phase 2: decision

- decision (commit/abort)
- OK

TC → A → B → C → D
Recovery in Two-Phase Commit

- Easy: just log the state-changes
  - Participants: prepared, uncertain, committed/aborted
  - Coordinator: prepared, committed/aborted, done
  - The messages are idempotent!
    - In recovery, resend whatever message was next
    - If coordinator and uncommitted: abort

- Two cases:
  - Recovery after timeouts
  - Recovery after crashes and reboots
  - (Note: you can’t differentiate between the above in a realistic, asynchronous network!)
Handling Timeouts

• Examples:
  Ex. 1: TC times out waiting for B’s vote
  Ex. 2: B times out waiting for TC’s decision message

• Btw, timeouts aren’t necessarily due to network
  – They could due to slow, overloaded hosts
Handling Timeouts on A/B/C/D

- TC times out waiting for B (or A/C/D)’s vote
- Can TC unilaterally decide to commit?
- Can TC unilaterally decide to abort?

Ex. 1
Handling Timeout on TC

• B times out waiting for TC’s decision

• If B voted “no” …
  – Can it unilaterally abort?

• If B responded with “yes” …
  – Can it unilaterally abort?
  – Can it unilaterally commit?
Termination Protocol

- If B times out on TC and has voted “yes”, then execute termination protocol:
  - B sends “status” message to A
    - If A has received “commit”/”abort” from TC, …
    - If A has not responded to TC, …
    - If A has responded with “no”, …
    - If A has responded with “yes”, …
Handling Crash and Reboot

- Nodes cannot back out if commit is decided

Examples:
- Ex 3: TC crashes just after deciding “commit”
  - Cannot forget about its decision after reboot
- Ex 4: A/B/C/D crashes after sending “yes”
  - Cannot forget about their response after reboot
Handling Crash and Reboot

- All nodes must log protocol progress
- What and when does TC log to disk?
- What and when does A/B/C/D log to disk?
Recovery Upon Reboot

• Ex 3: TC crashes:
  – If TC finds no “commit” on disk, abort
  – If TC finds “commit”, commit

• Ex 4: A/B/C/D crash:
  – If A/B/C/D finds no “yes” on disk, abort
  – If A/B/C/D finds “yes”, run termination protocol to decide
Fault-Tolerance Limitations of Recovery-enabled 2PC

• Even with recovery enabled, 2PC isn’t really fault-tolerant (or live), because it can block even when one (or a few) machines fail
  – Blocking means that it doesn’t make progress during the failure

• Can you think of an example fault scenario?
Example Blocking Failure for 2PC

- Scenario:
  - TC sends commit outcome to A, A gets it and commits, and then both TC and A die
  - B, C, D have already also replied Yes, have locked their mutexes, and now need to wait for TC or A to reappear
    - They cannot recover the decision with certainty until TC or A are online
  - If that takes a long time (e.g., a human needs to replace a hardware component), then the protocol is stuck and availability goes down
  - If TC is also participant, as it typically is, then this protocol is vulnerable to a single-node failure (the TC’s failure)!

- This is why 2 phase commit is called a blocking protocol
  - Btw, the original, non-recovery-enabled protocol blocked even more frequently, but we’ve fixed some of the obvious glitches

- In context of consensus requirements: 2PC is safe, but not live
Fixing Two-Phase Commit

• Surprisingly enough, there’s no simple fix!
  – Creating a protocol that’s both correct and available is tough!
  – In fact, as we’ll see at the end of the class, it’s impossible in the general sense (and it can be proven so!!)
  – But it’s tough to even get close to that

• It took 25 years to come up with safe protocol
  – 2PC appeared in 1979 (Gray)
  – In 1981, a basic, unsafe 3PC was proposed (Stonebraker)
  – In 1998, the safe, mostly live Paxos appeared (Lamport)
  – Why so difficult? Well, we’ll see later…
Next Time

- Three Phase Commit
- Paxos
- Usage of them
Extra Readings

• Two-phase commit:

• Three-phase commit:
  – http://the-paper-trail.org/blog/consensus-protocols-three-phase-commit

• Paxos:
  – http://the-paper-trail.org/blog/consensus-protocols-paxos/

• FLP impossibility result in distributed systems: