Distributed Systems

Coordination

Dynamic and Distributed Information Systems

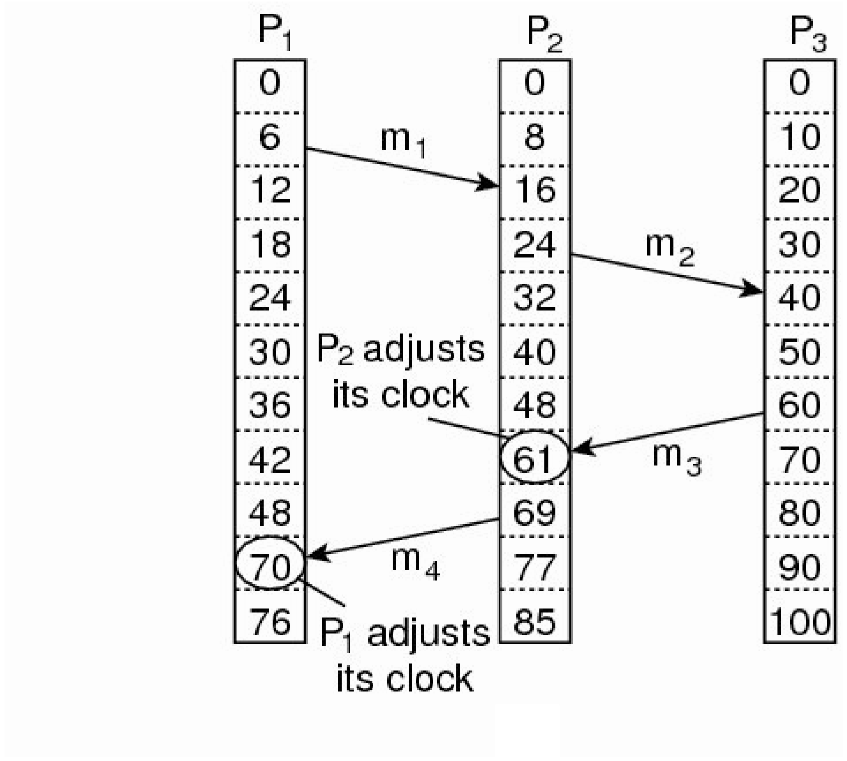
Today's Agenda

- Vector Clocks
- Atomicity
  - Two-Phase Commit

Vector Clocks

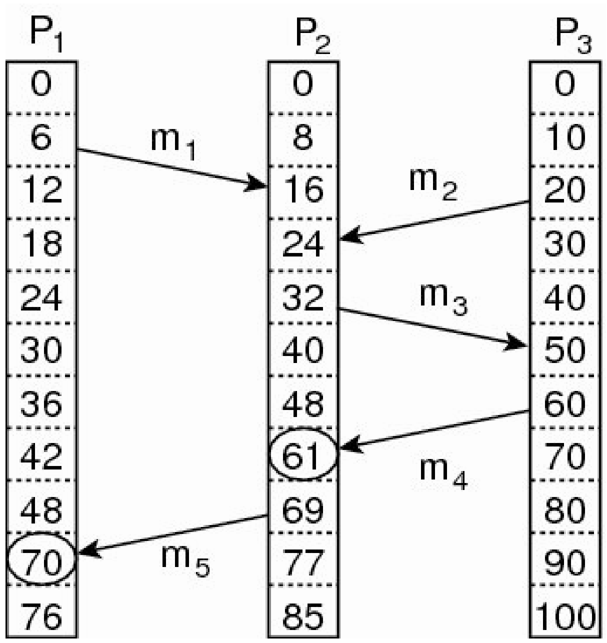
Dynamic and Distributed Information Systems

Logical Clocks



- Lamport's Timestamps can be used for total ordering of events
  - However, the notion of causality (dependencies between events) is lost
  - Also, total ordering is often too strict

Example



- Example 1: The reception of m3 (50) could depend on the reception of m2 (24) and m1 (16). That’s correct.
- Example 2: The sending of m2 (20) seems to be dependent on the reception of m1 (16)
  - But is it? No!

Causality

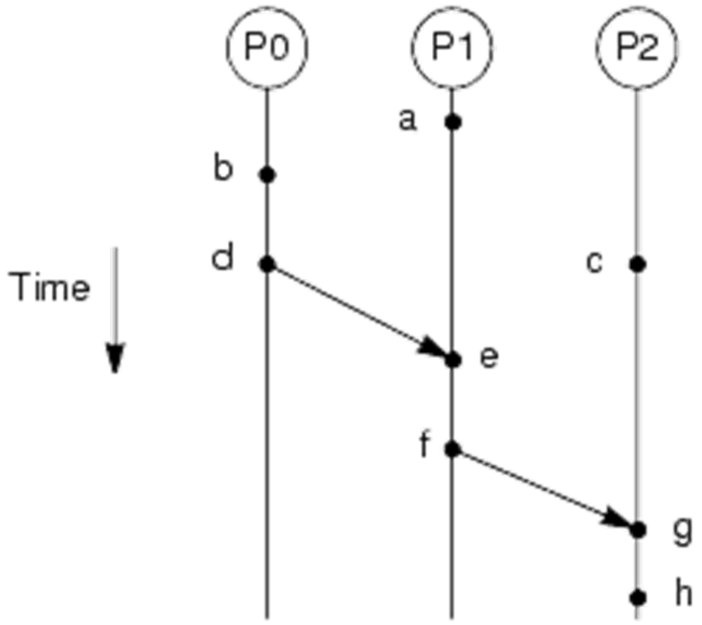


Figure 3-3. Events in a distributed system

- Generally, two events can:
  - Be linked by a dependency (*a* → *b*, which means *a happens before b*)
    - E.g., *b*→*d, d*→*e, b*→*e, b*→*h*
  - Independent (concurrent)
    - E.g., *a* and *c*, or *a* and *b*

Inefficiency of Logical Clocks

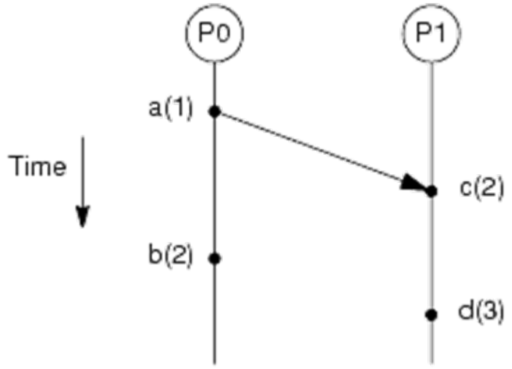


Figure 3-4. Totally ordered logical clock timestamps

- If we want to observe causality with Logical Clocks (a.k.a. Lamport Timestamps), we may fail:
  - *d* consistently has a later timestamp than *b*, so we would (wrongly) assume that *b*→*d*

Vector Clocks

- We have *N* nodes
  - Each node maintains a vector of *N* logical clocks
  - One logical clock is its own
  - The rest *N*-1 logical clocks are estimations for the other nodes
- Logical clocks are managed as follows:
  - They are all initialized with zero
  - When an even happens in a node, it increases it own logical clock in the vector by one
  - When a node sends a message, it includes its whole vector
  - When a node receives a message, it updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element)
- An event *a* is considered to happen before event *b*, only if all elements of the VC of *a* are less than or equal that the respective elements of the VC of *b*. (in fact, at least one element has to be *lower*)

Vector Clocks

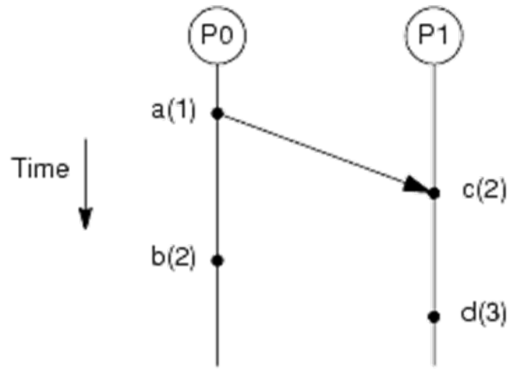


Figure 3-4. Totally ordered logical clock timestamps

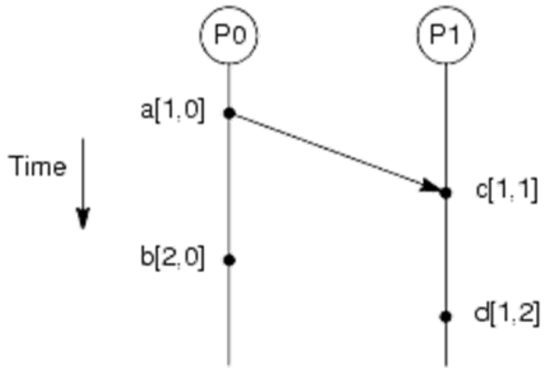


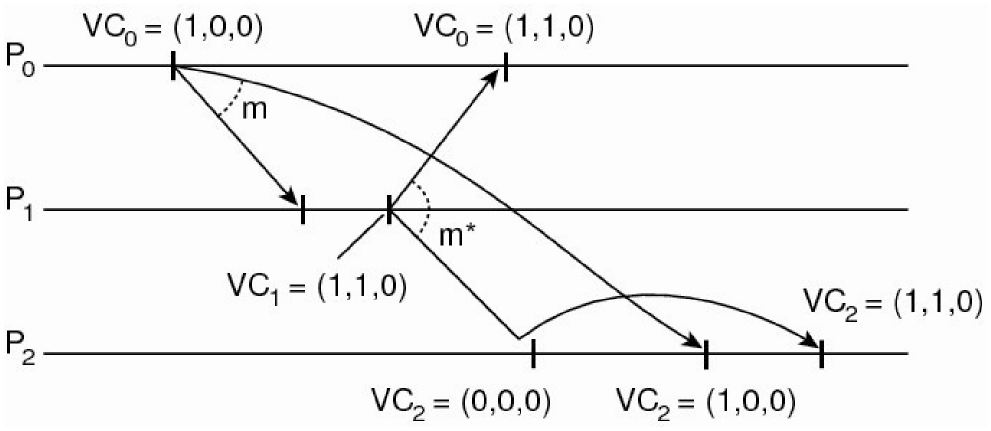
Figure 3-5. Partially ordered logical clock timestamps

Logical Clocks **Vector Clocks**

- With Vector Clocks, we can see that
  - *a* → *c*, because [1,0] < [1,1]
  - *a* → *d*, because [1,0] < [1,2]
  - Same for *a*→*b*, *c*→*d*
  - But *b* and *d* are *independent* (*concurrent*), because there is no clear order between [2,0] and [1,2]

Causal Communication

- Vector Clocks can be used to enforce causal communication
  - Do not deliver a packet until all causally earlier packets have been delivered
- Assumptions
  - No packets get lost
  - Clocks are increased only when sending a new message
  - A packet is delivered when only the sender’s logical clock is increased



Causal Ordering: at which layer?

- Middleware layer:
  - +: Generic approach
  - - : Potential (but not definite) causality is captured
    - Even messages that are not related, but happen to occur in a given order, are assumed dependent: this makes it “heavier” than necessary
  - -: Some causality may not be captured
    - Alice posts a message, and then calls Bob and informs him about that message. Bob may take some other action that *depends* on the information he got from Alice, even *before* receiving the official message of Alice. This causality is not captured by the middleware.
    - Generally, external communication can mess up the assumptions of the middleware
- Application-specific Causal Ordering
  - +: Can be tuned to be more lightweight
  - +: Can be tuned to be more accurate
  - -: Puts the burden of causality checking on the application developer

Atomicity

Dynamic and Distributed Information Systems

Atomicity: The Issue

- Consider a replicated database
  - equipped with a DS that guarantees reliable multicasting
  - updates are multicast to all replicas, and the system guarantees that they are delivered in order
- Is this enough???
- Nasty scenario:
  - A message is multicast reliably to all replicas, and is delivered to the application layer (the database)
  - One replica crashes while performing the update
  - When it recovers it is at an inconsistent state
- Atomicity is what we need
  - All commit, or all abort!
  - Guarantee that an operation is completed at all participants (or at none of them)

Example

- Transfer money from bank A to bank B
  - Debit A, credit B, tell client “OK”
- We want either both to do it or neither to do it
  - Never want only one side to act
  - Better if nothing happens!
- Goal: Atomic Commit Protocol

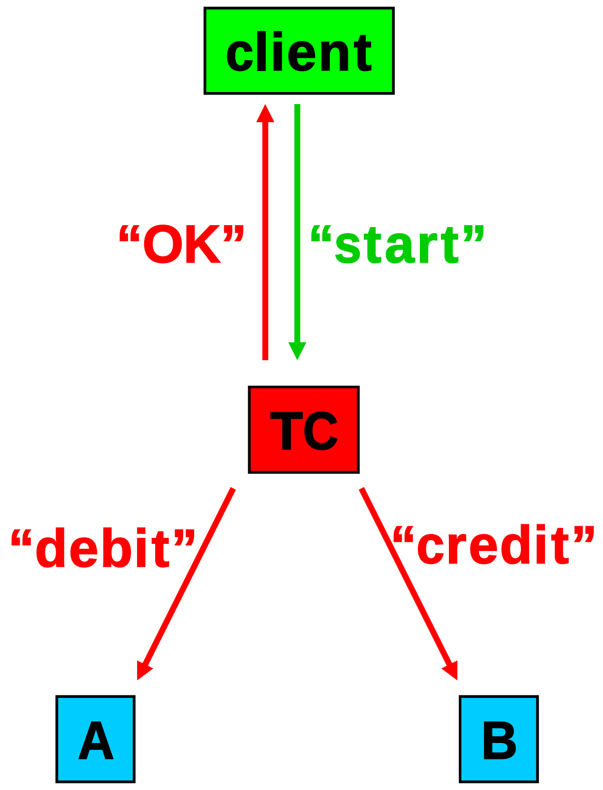
Two Kinds of Atomicity

- 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"
  - No partial results
- For serializability we use synchronization (logical / vector clocks)
- Now we are going to deal with recoverability

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!

One-Phase Commit (1PC)



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

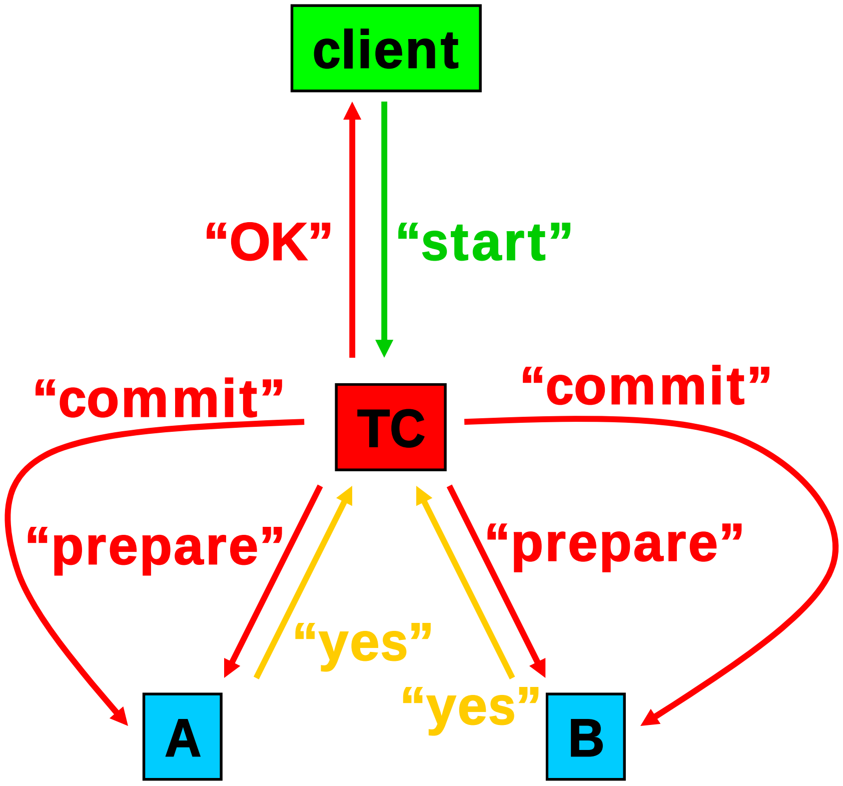
Failure Scenarios

- 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
- One network link (of A or B) is broken
  - One commits, the other doesn't
- One of A or B has crashed
  - One commits, the other doesn't
- TC crashes between sending to A and B
  - A commits, B doesn't

Atomic Commit: Desirable Properties

- TC, A, and B have separate notions of committing
- Correctness
  - 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

Two-Phase Commit (2PC)



- Same entities as in 1PC
- Operation
  - 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.

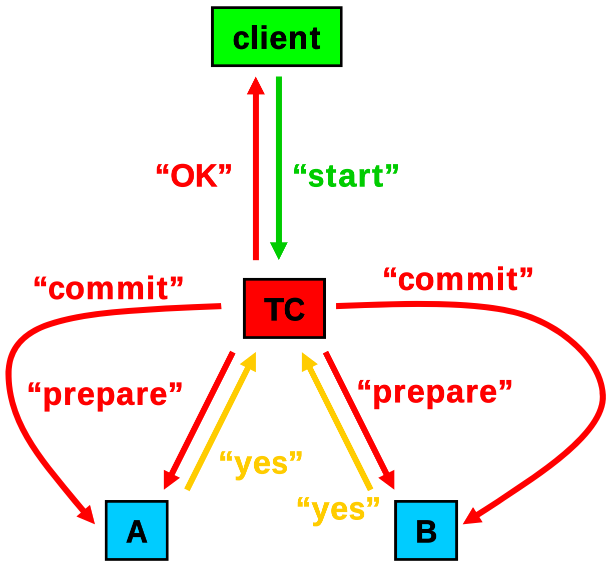
2PC: Correctness, Liveness?

- 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 get lost?

2PC: Liveness Problems

- Timeout
  - Host is up, but doesn't receive message it expects
  - Maybe other host crashed, maybe network dropped message, maybe network is down1
  - 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 Liveness Problems



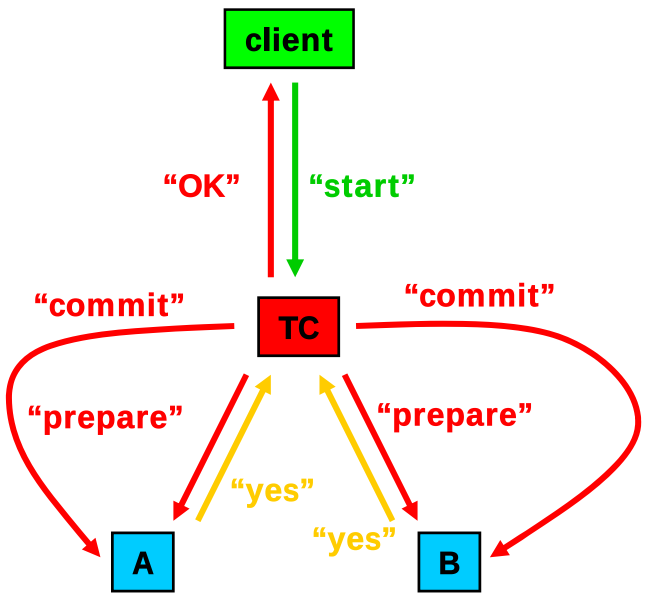
- Solution
  - Introduce timeouts
  - Take appropriate actions
    - but be conservative to preserve correctness!

- Where in the protocol do hosts wait for messages?
  - TC waits for "yes"/"no" from A and B
  - A and B wait for "commit"/"abort" from TC

When TC times out

- Proceed with a decision when TC waits too long for *yes/no*
- TC has not yet sent any "commit" messages
  - So it can safely abort: sends "abort" messages
- This preserves safety, but sacrifices liveness (why?)
  - Perhaps both A, B prepared to commit, but a "yes" message was lost
  - Could have committed, but TC unaware!
  - Thus, TC is conservative

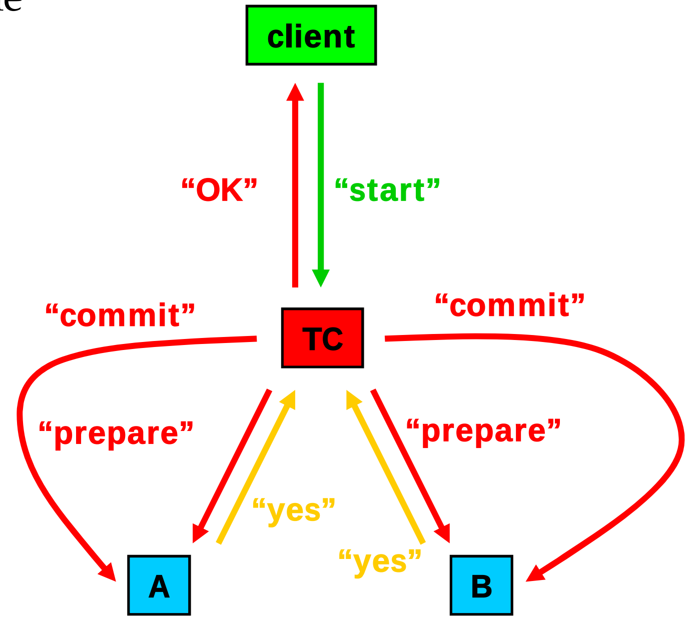
When B (or A) times out



- If B voted "no"
  - It can unilaterally abort
  - The TC will never send "commit" in this case
- If B voted "yes"
  - Can it unilaterally abort?
    - No!!
    - 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"
- So, what do we do if B voted "yes"???
  - Either B keeps waiting forever (not a solution)
  - Or we devise a better plan pull it out of indefinite waiting, without hurting correctness

Termination Protocol if B voted "yes"

- B has voted "yes", but is waiting for an answer too long
- B directly contacts A: sends "status" request to A, asking if A knows whether the transaction should commit
  - If A received "commit" or "abort" from TC: B decides same way (can't disagree with TC)
  - If A hasn't voted anything yet: B and A both abort
    - TC can't have decided "commit"; 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, keep waiting
    - TC might have decided "commit" and replied to client
    - TC might have timed out, aborted, and replied to client
  - If no reply from A: no decision possible, wait for TC



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!

Solution: Persistent State

- Storing state in non-volatile memory (e.g., a disk)
  - If all nodes know their pre-crash state, they can use the previously described termination protocol
  - A and B can also ask TC, which may still remember if it committed
- In what order do we store & send?
  - write disk, then send “yes” message if A/B, or “commit” if TC?
  - or vice-versa?
- Can we 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”
- So, should 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

- 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

2PC: 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 occur and all say "yes," then commit
  - Liveness: if failures occur, then repair, wait long enough, eventually take some decision

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)