Two-Phase Commit

COS 418: Distributed Systems
Lecture 6

Daniel Suo

Acknowledgements: Kyle Jamieson, Mike Freedman, Irene Zhang

Plan

- Fault tolerance in a nutshell
- Safety and liveness
- Two-phase commit

Fault tolerance in a nutshell

What is fault tolerance?

- Building **reliable** systems from **unreliable** components

- Three basic steps
  1. **Detecting errors**: discovering the presence of an error in a data value or control signal
  2. **Containing errors**: limiting how far errors propagate
  3. **Masking errors**: designing mechanisms to ensure a system operates correctly despite error (and possible correct error)

Why is fault tolerance hard?

Failures Propagate

- Say **one bit** in a DRAM fails…
- …it **flips a bit** in a memory address the kernel is writing to...
- ...causes big memory error elsewhere, or a **kernel panic**...
- ...program is running one of many distributed file system storage servers...
- ...a client **can't read from FS**, so it hangs.

So what to do?

1. **Do nothing**: silently return the failure
2. **Fail fast**: detect the failure and report at interface
   - Ethernet station jams medium on detecting collision
3. **Fail safe**: transform incorrect behavior or values into acceptable ones
   - Failed traffic light controller switches to blinking-red
4. **Mask the failure**: operate despite failure
   - Retry op for transient errors, use error-correcting code for bit flips, replicate data in multiple places

Masking failures

- We mask failures on **one server** via
  - Atomic operations
  - Logging and recovery

- In a distributed system with **multiple servers**, we might replicate some or all servers

- But if you give a mouse some replicated servers
  - She's going to want a way to keep them all consistent in a fault tolerant way

Safety and liveness

Reasoning about fault tolerance

- This is hard!
  - How do we design fault-tolerant systems?
  - How do we know if we're successful?
- Often use "properties" that hold true for every possible execution
- We focus on **safety** and **liveness** properties

Safety

- "Bad things" don't happen
  - No stopped or deadlocked states
  - No error states
- Examples
  - Mutual exclusion: two processes can't be in a critical section at the same time
  - Bounded overtaking: if process 1 wants to enter a critical section, process 2 can enter at most once before process 1

Liveness

- “Good things” happen
  - …eventually
- Examples
  - Starvation freedom: process 1 can eventually enter a critical section as long as process 2 terminates
  - Eventual consistency: if a value in an application doesn’t change, two servers will eventually agree on its value

Often a tradeoff

- “Good” and “bad” are application-specific
- Safety is very important in banking transactions
  - May take some time to confirm a transaction
- Liveness is very important in social networking sites
  - See updates right away (what about the “breakup problem”?)

Two-phase commit

Motivation: sending money

```
send_money(A, B, amount) {
   Begin_Transaction();
   if (A.balance - amount >= 0) {
       A.balance = A.balance - amount;
       B.balance = B.balance + amount;
       Commit_Transaction();
   } else {
       Abort_Transaction();
   }
}
```

Single-server: ACID

- **Atomicity**: all parts of the transaction execute or none (A's decreases and B's balance increases)
- **Consistency**: the transaction only commits if it preserves invariants (A's balance never goes below 0)
- **Isolation**: the transaction executes as if it executed by itself (even if C is accessing A's account, that will not interfere with this transaction)
- **Durability**: the transaction's effects are not lost after it executes (updates to the balances will remain forever)

Distributed transactions?

- Partition databases across multiple machines for scalability (A and B might not share a server)
- A transaction might touch more than one partition
- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?

Two-Phase Commit (2PC)

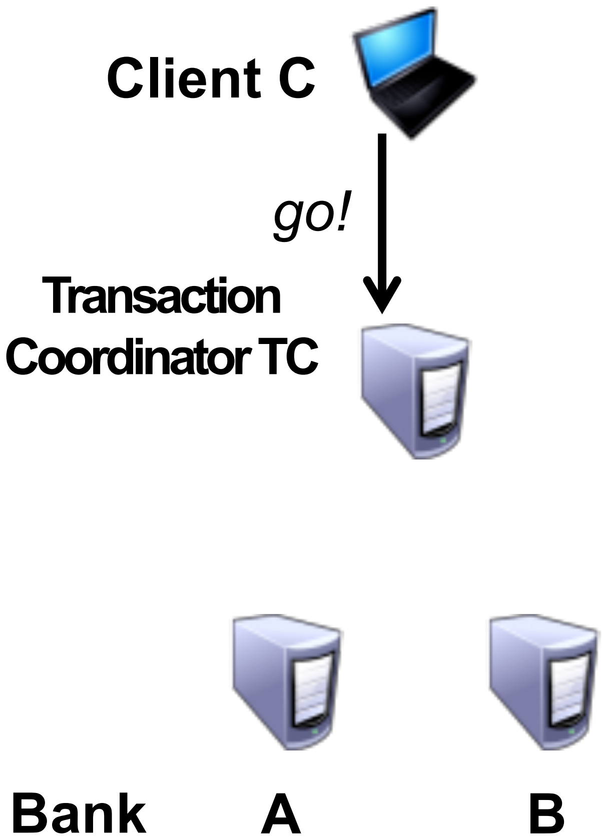
- **Goal:** General purpose, distributed agreement on some action, with failures
  - Different entities play different roles in the action

- **Running example:** Transfer money from A to B
  - Debit at A, credit at B, tell the client "okay"
  - Require **both** banks to do it, or **neither**
  - Require that **one bank never act alone**

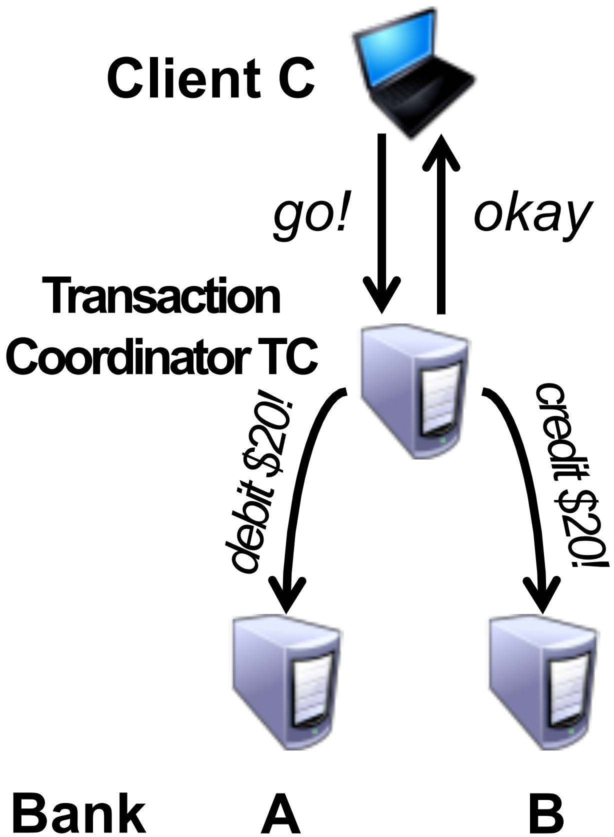
- This is an **all-or-nothing** atomic commit protocol
  - Later will discuss how to make it **before-or-after** atomic

Straw Man protocol

1. **C → TC**: *"go!"*



Straw Man protocol



1. **C → TC**: *"go!"*

2. **TC → A**: *"debit $20!"*
   TC → B: *"credit $20!"*
   TC → C: *"okay"*

- **A, B** perform actions on receipt of messages

Reasoning about the Straw Man protocol

What could **possibly** go wrong?

1. Not enough money in **A's** bank account?
2. **B's** bank account no longer exists?
3. **A** or **B** **crashes** before receiving message?
4. The best-effort network to **B** **fails**?
5. **TC** **crashes** after it sends *debit* to **A** but before sending to **B**?

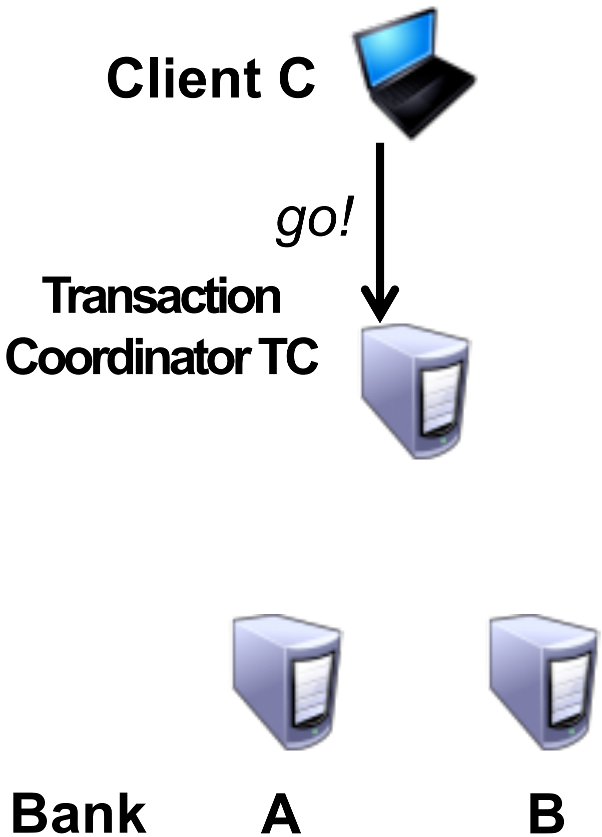
Safety versus liveness

- Note that **TC**, **A**, and **B** each have a notion of committing
- We want two properties:

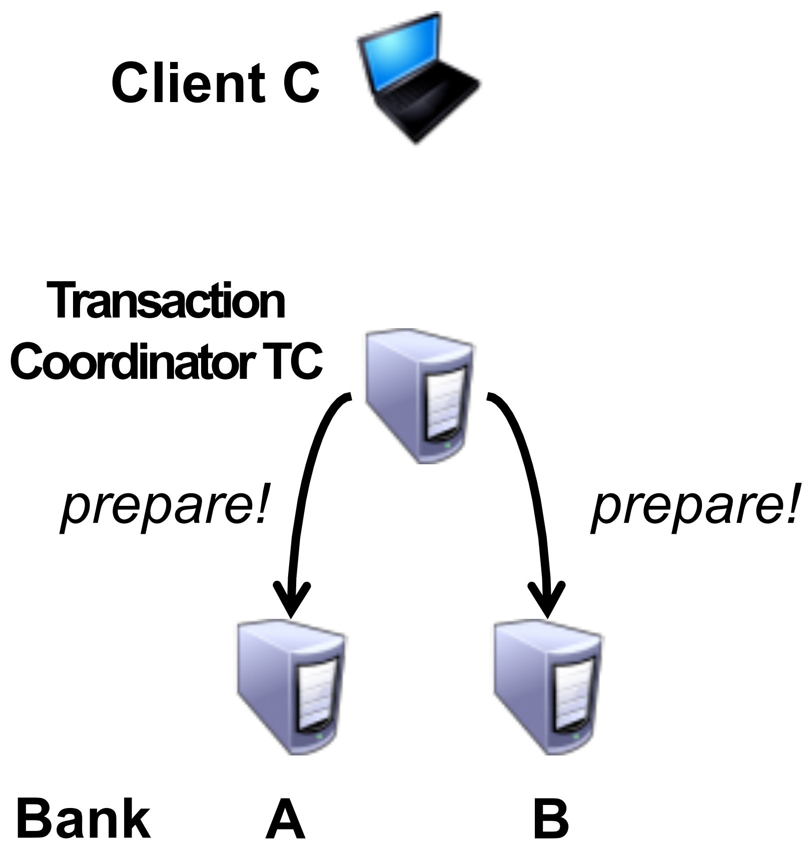
1. Safety
   - If **one commits, no one aborts**
   - If **one aborts, no one commits**
2. Liveness
   - If **no failures** and **A** and **B** can commit, **action commits**
   - If **failures,** reach a conclusion ASAP

A *correct* atomic commit protocol

1. C → TC: *"go!"*

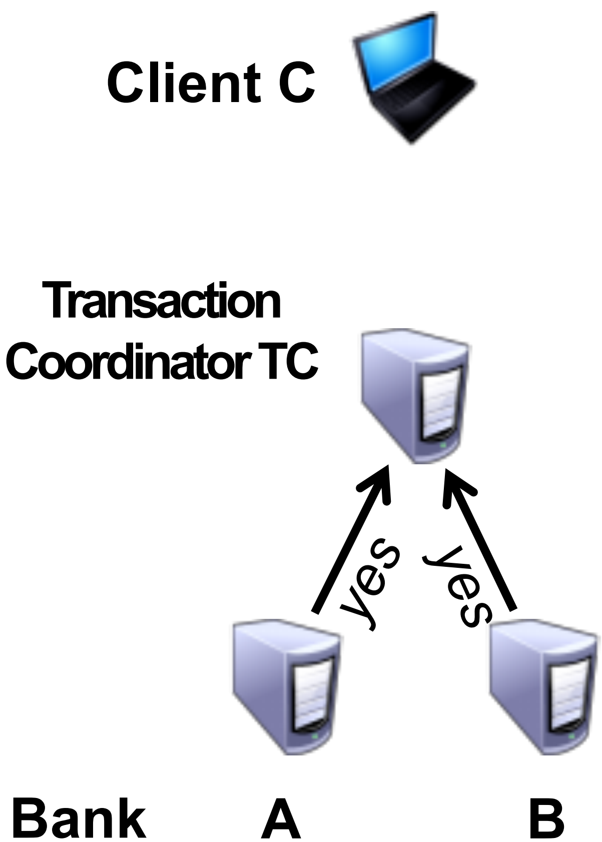


A *correct* atomic commit protocol



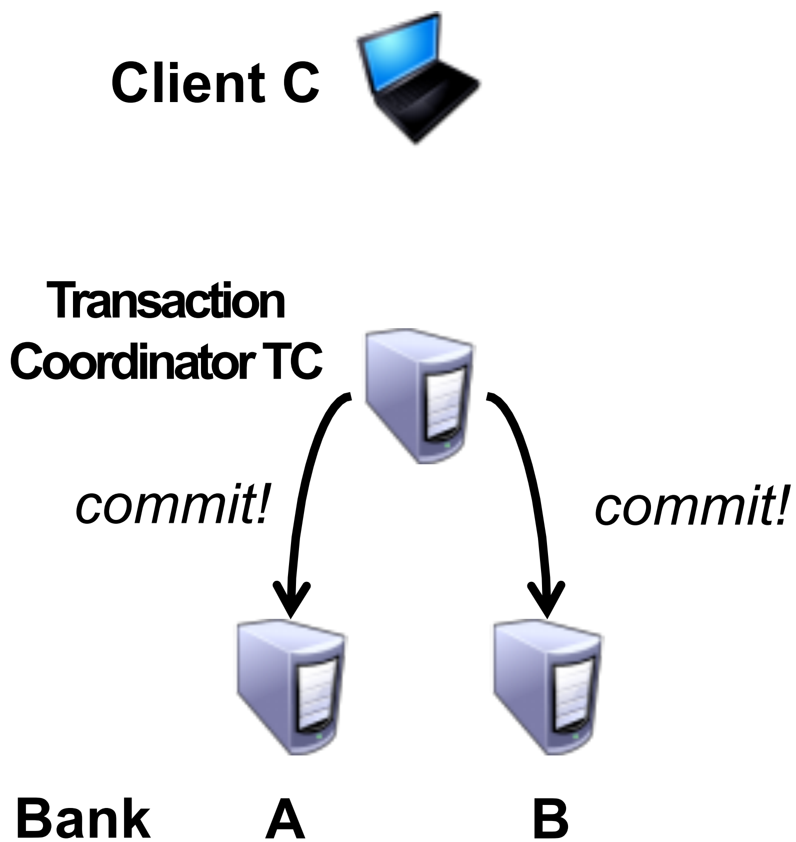
1. C → TC: *"go!"*
2. TC → A, B: *"prepare!"*

A *correct* atomic commit protocol



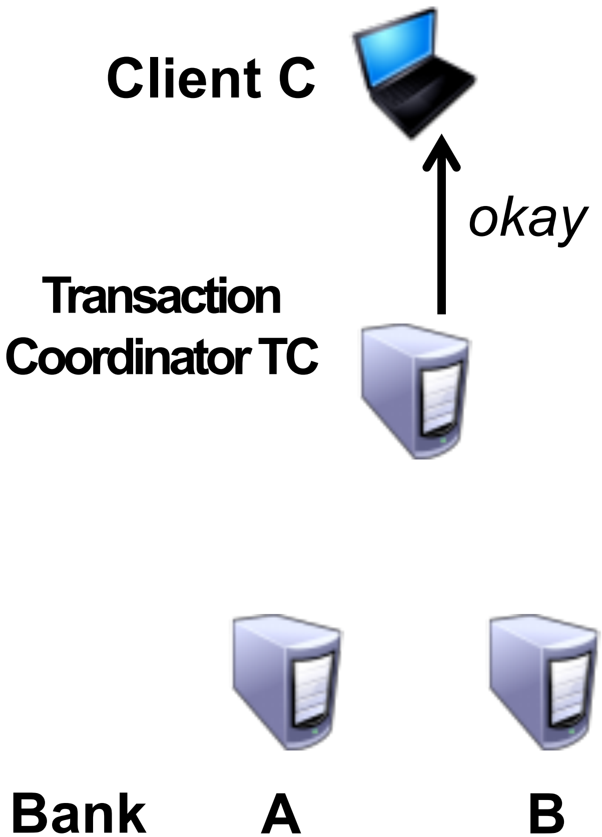
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   - **TC** sends ***commit*** if **both** say *yes*
   - **TC** sends ***abort*** if **either** say *no*

A *correct* atomic commit protocol



1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*
4. **TC → A, B:** *"commit!"* or *"abort!"*
   - **TC** sends ***commit*** if **both** say *yes*
   - **TC** sends ***abort*** if **either** say *no*
5. **TC → C:** *"okay" or "failed"*

- **A, B** commit on receipt of commit message

Reasoning about atomic commit

- *Why is this correct?*
  - Neither can commit unless both agreed to commit

- *What about performance?*

  1. **Timeout:** I'm up, but didn't receive a message I expected
     - Maybe other node crashed, maybe network broken
  2. **Reboot:** Node crashed, is rebooting, must clean up

Timeouts in atomic commit

Where do hosts **wait** for messages?

1. **TC** waits for "yes" or "no" from **A** and **B**
   - **TC** hasn't yet sent any commit messages, so can **safely abort** after a timeout
   - But this is **conservative:** might be network problem
     - We've preserved correctness, sacrificed performance

2. **A** and **B** wait for "commit" or "abort" from **TC**
   - If it sent a *no*, it can **safely abort** *(why?)*
   - If it sent a *yes*, can it unilaterally abort?
   - Can it unilaterally commit?
   - A, B could wait forever, but there is an alternative…

Server termination protocol

- Consider Server **B** (Server **A** case is symmetric) waiting for *commit* or *abort* from **TC**
  - Assume **B** voted *yes* (else, unilateral abort possible)

- **B → A:** "status?" **A** then replies back to **B.** Four cases:
  - (No reply from **A**): no decision, **B** waits for **TC**
  - Server **A** received commit or abort from **TC:** Agree with the **TC's** decision
  - Server **A** hasn't voted yet or voted *no:* both **abort**
    - **TC** can't have decided to commit
  - Server **A** voted *yes:* both must **wait** for the **TC**
    - **TC** decided to **commit** if both replies received
    - **TC** decided to **abort** if it timed out

Reasoning about the server termination protocol

- *What are the liveness and safety properties?*
  - **Safety**: if servers don't crash, all processes will reach the same decision
  - **Liveness**: if failures are eventually repaired, then every participant will eventually reach a decision
- Can resolve **some** timeout situations with guaranteed correctness
- Sometimes however **A** and **B** must block
  - Due to failure of the **TC** or network to the **TC**
- But what will happen if **TC, A,** or **B** **crash and reboot?**

How to handle crash and reboot?

- Can't back out of commit if already decided
  - **TC** crashes just after sending *"commit!"*
  - **A** or **B** crash just after sending *"yes"*
- If all nodes knew their state before crash, we could use the termination protocol…
  - Use **write-ahead log** to record *"commit!" and "yes"* to disk

Recovery protocol with non-volatile state

- If everyone rebooted and is reachable, TC can just check for **commit** record on disk and **resend** action
- **TC:** If no **commit** record on disk, **abort**
  - You didn't send any *"commit!"* messages
- **A, B:** If no **yes** record on disk, **abort**
  - You didn't vote *"yes"* so **TC** couldn't have committed
- A, B: If **yes** record on disk, execute termination protocol
  - This might block

Two-Phase Commit

- This recovery protocol with non-volatile logging is called ***Two-Phase Commit (2PC)***

- **Safety:** All hosts that decide reach the same decision
  – No commit unless everyone says "yes"

- **Liveness:** If no failures and all say "yes" then commit
  – **But if failures then 2PC might block**
  – **TC must be up to decide**

- **Doesn't tolerate faults well: must wait for repair**

Assignment 2

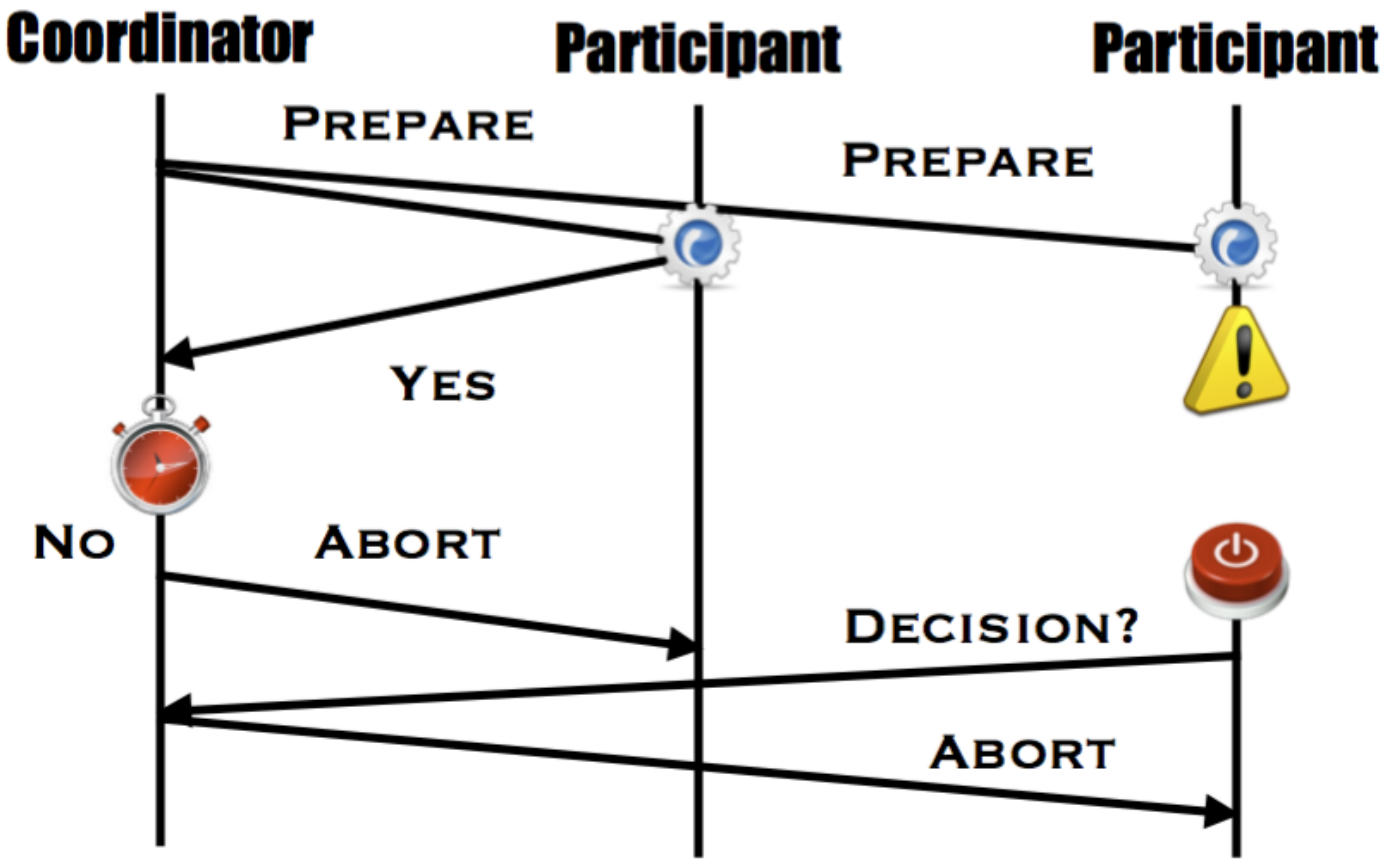
Due October 19

Wednesday topic

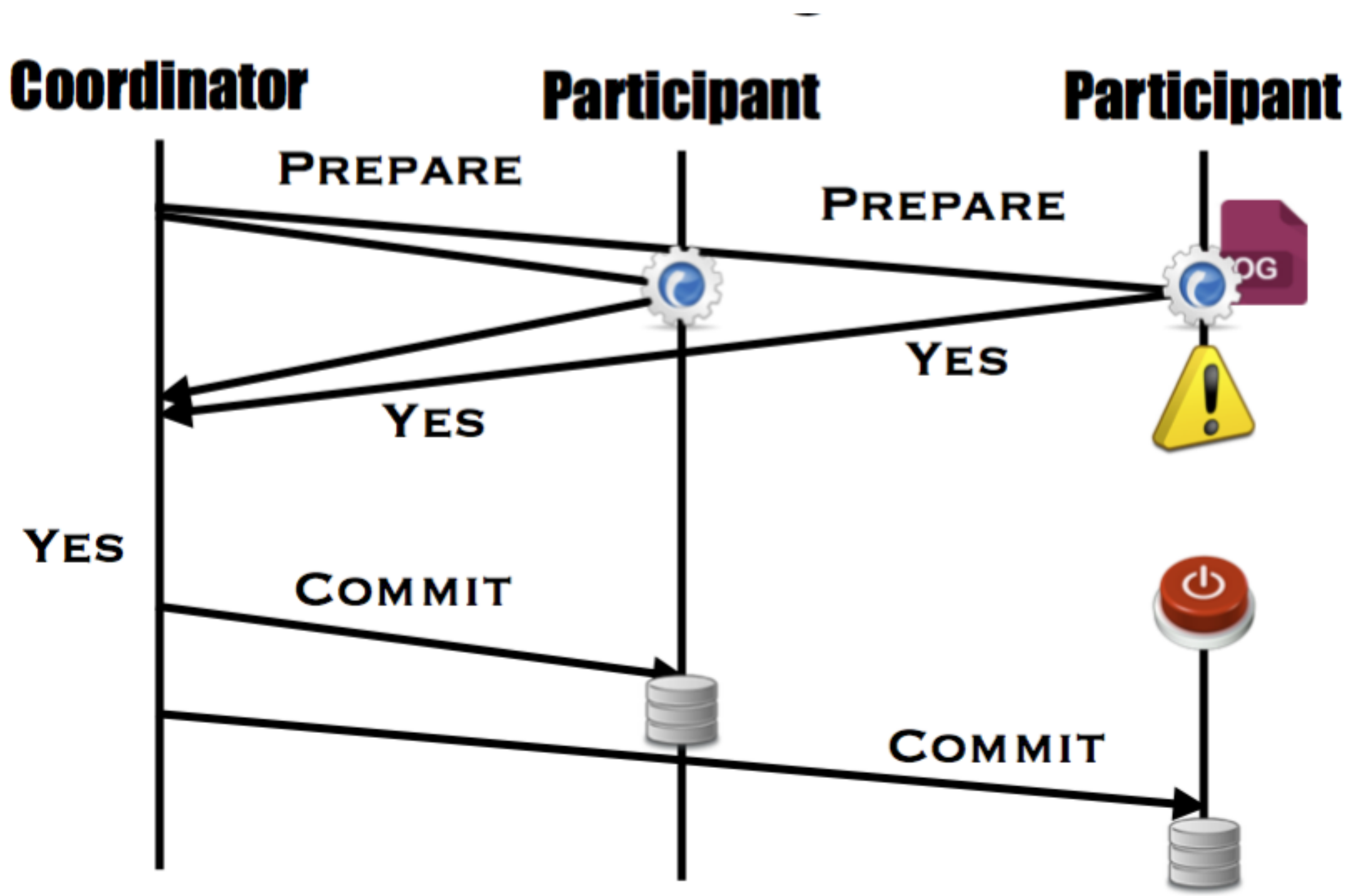
Consensus I: FLP Impossibility and Paxos

Two-phase commit failure scenarios

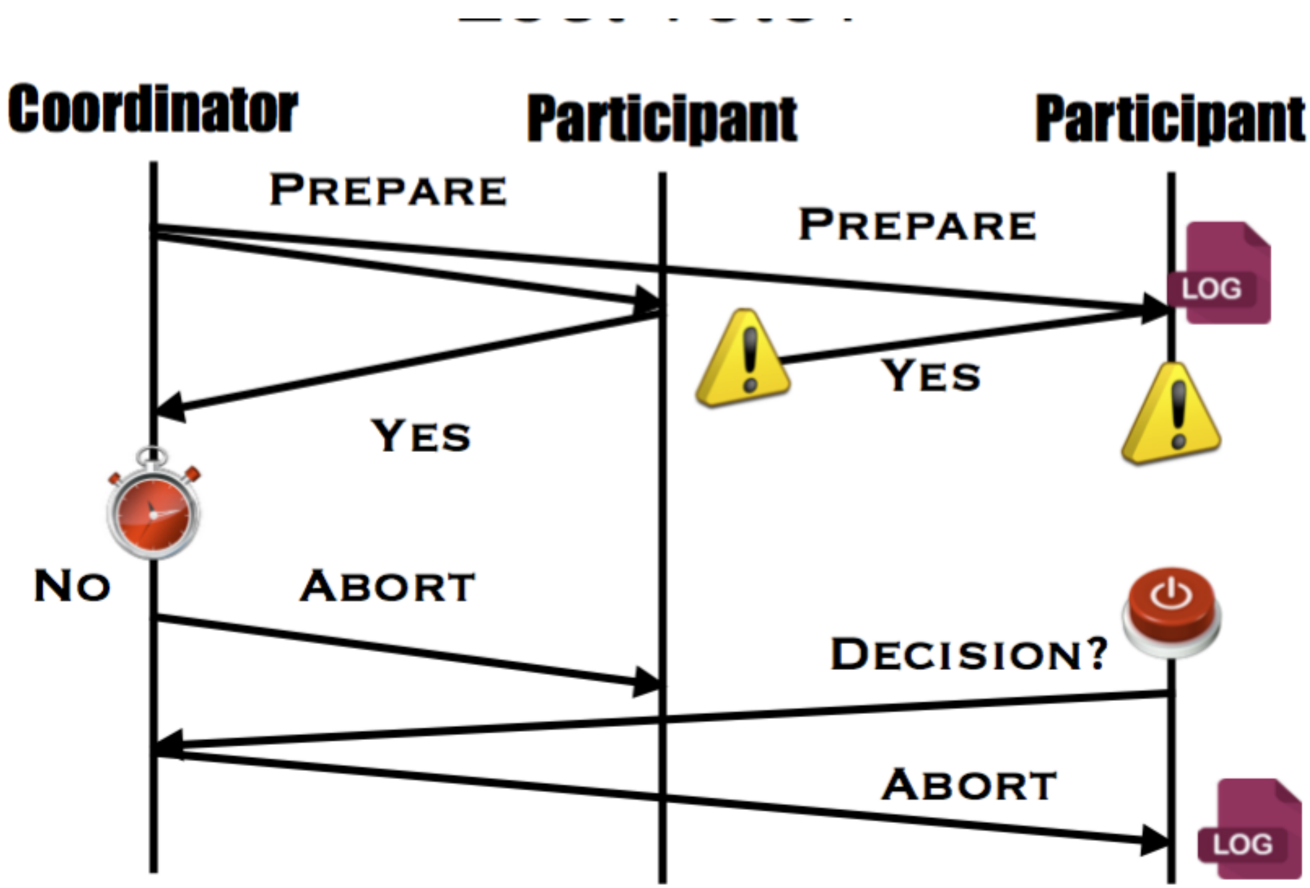
What if participant fails before sending response?



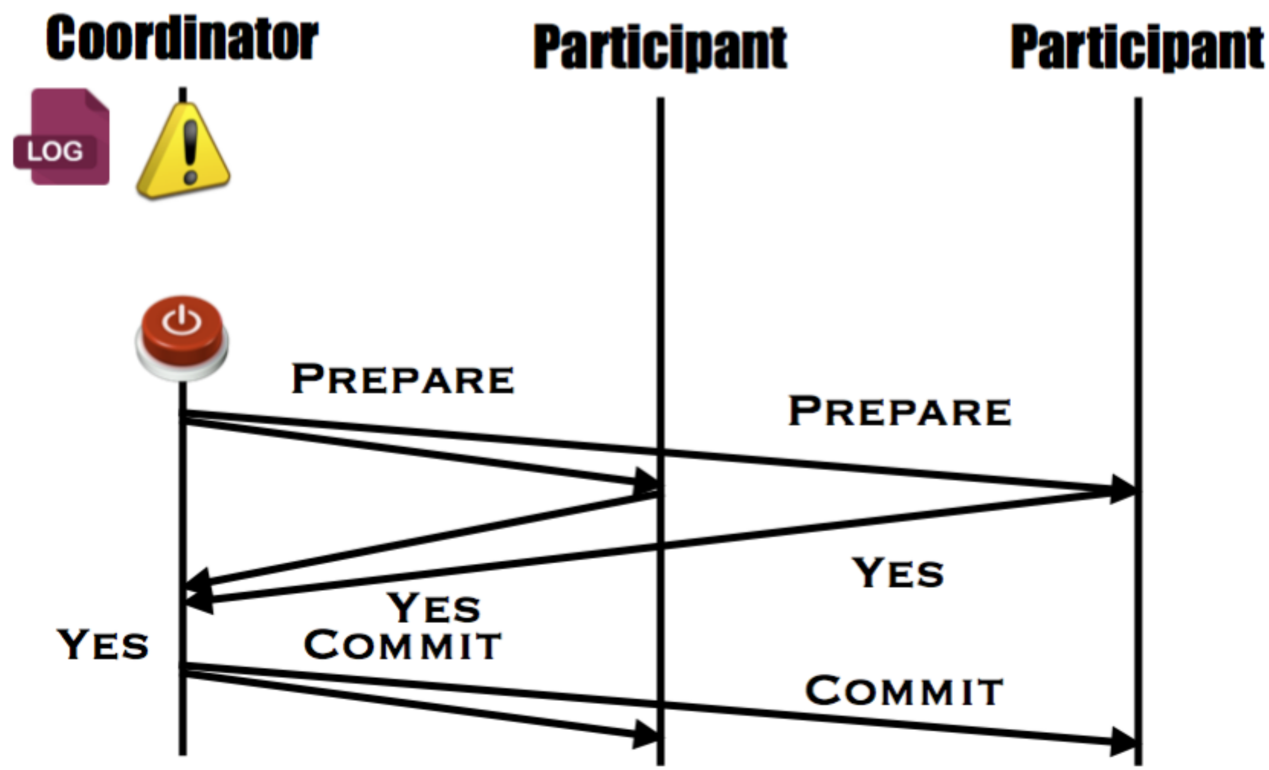
What if participant fails after sending vote



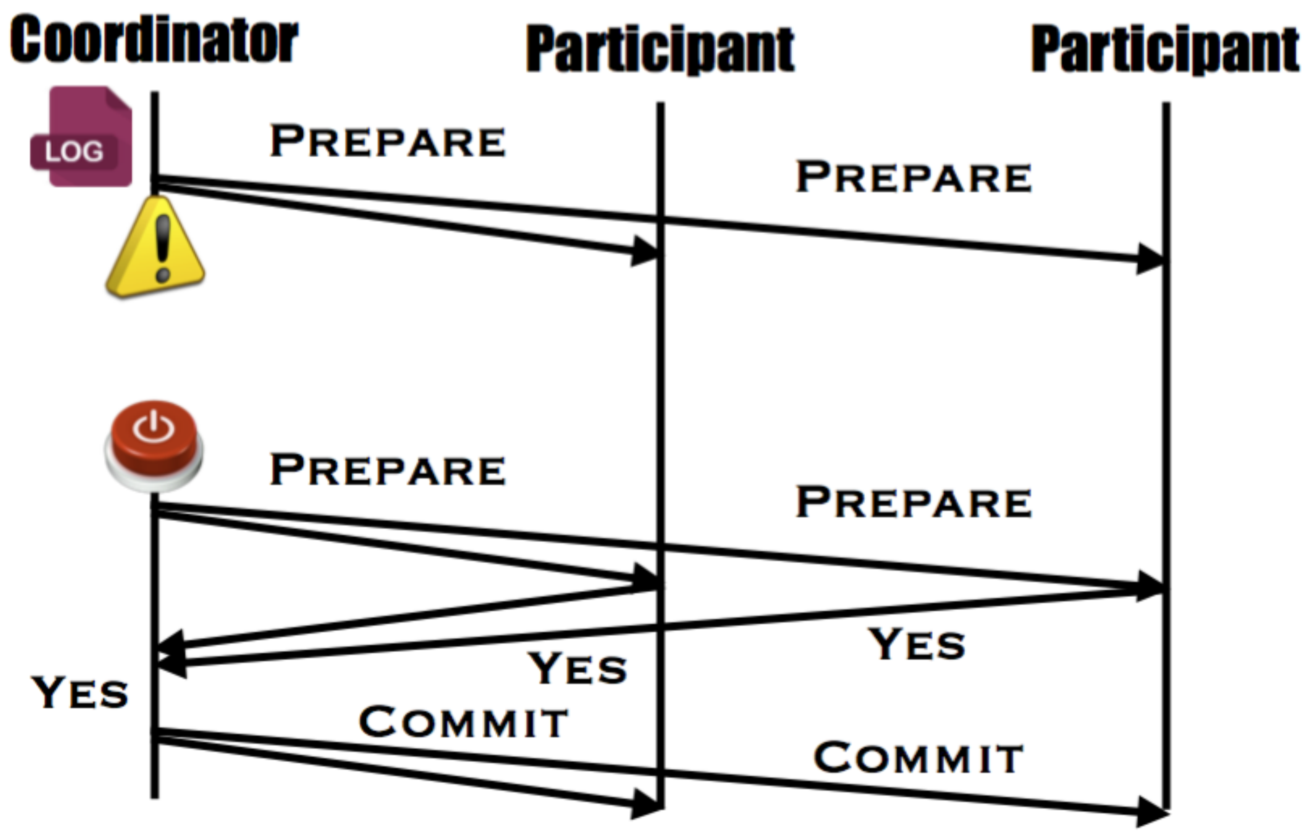
What if participant lost a vote?



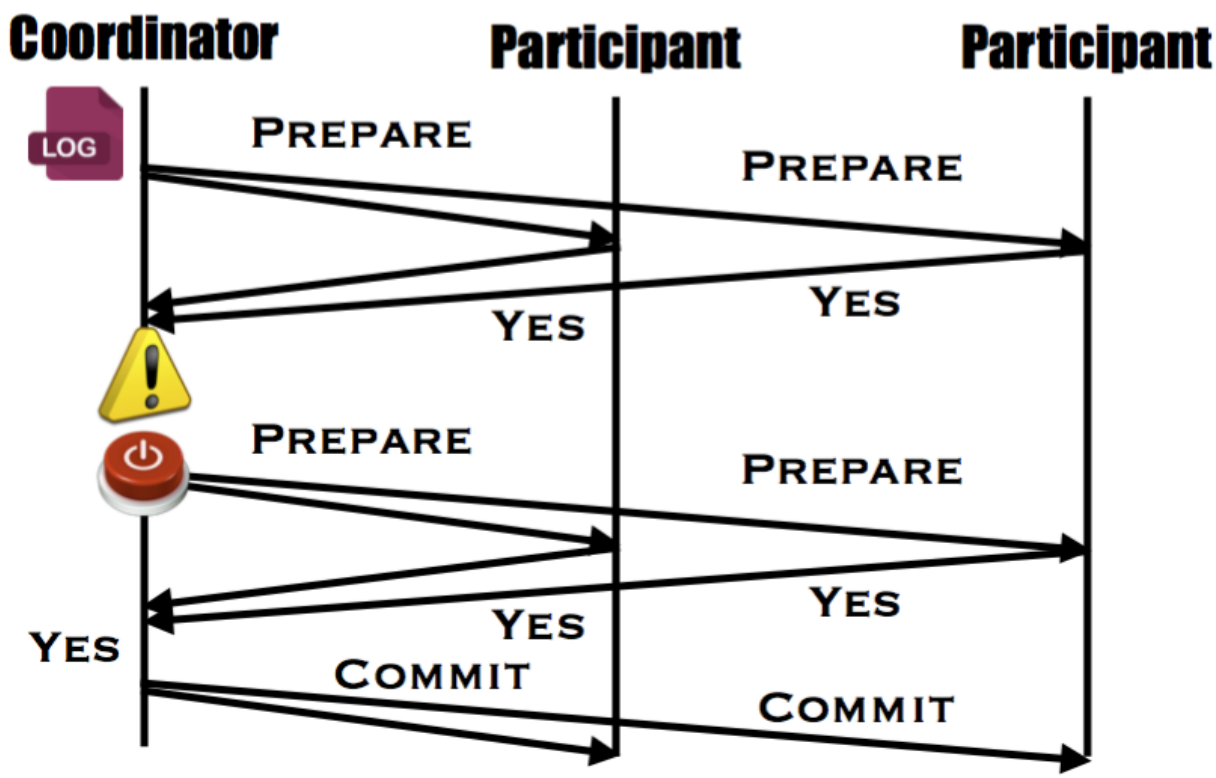
What if coordinator fails before sending prepare?



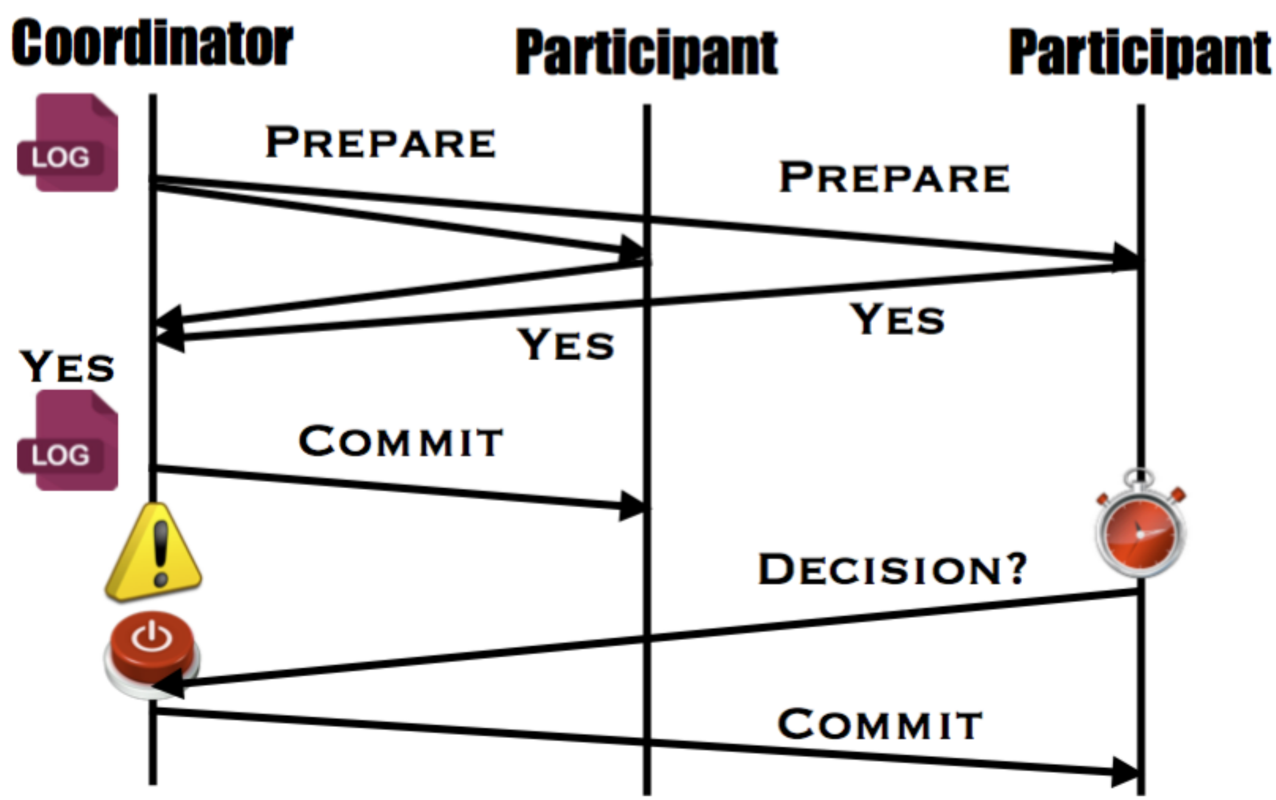
What if coordinator fails after sending prepare?



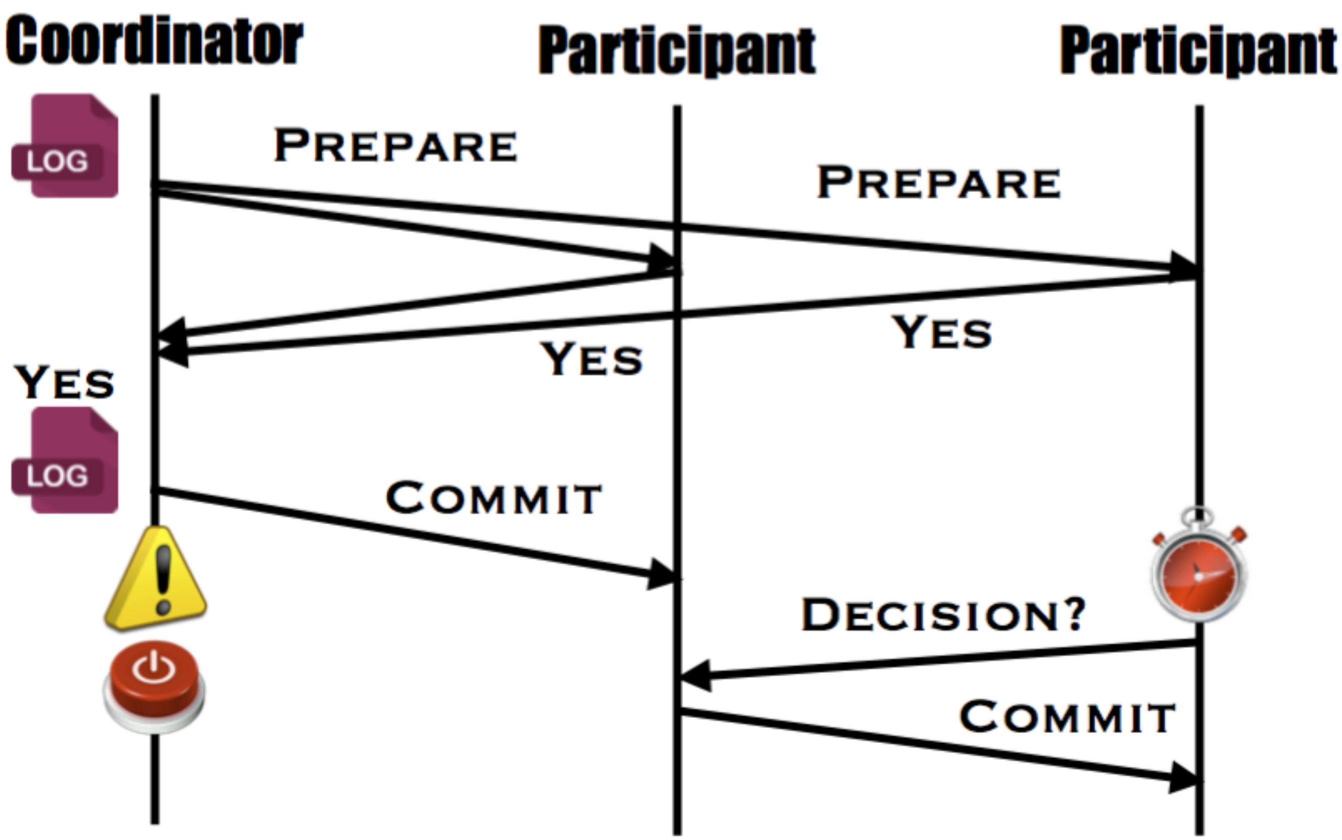
What if coordinator fails after receiving votes



What if coordinator fails after sending decision?



Do we need the coordinator?



What happens if we don't have a decision?