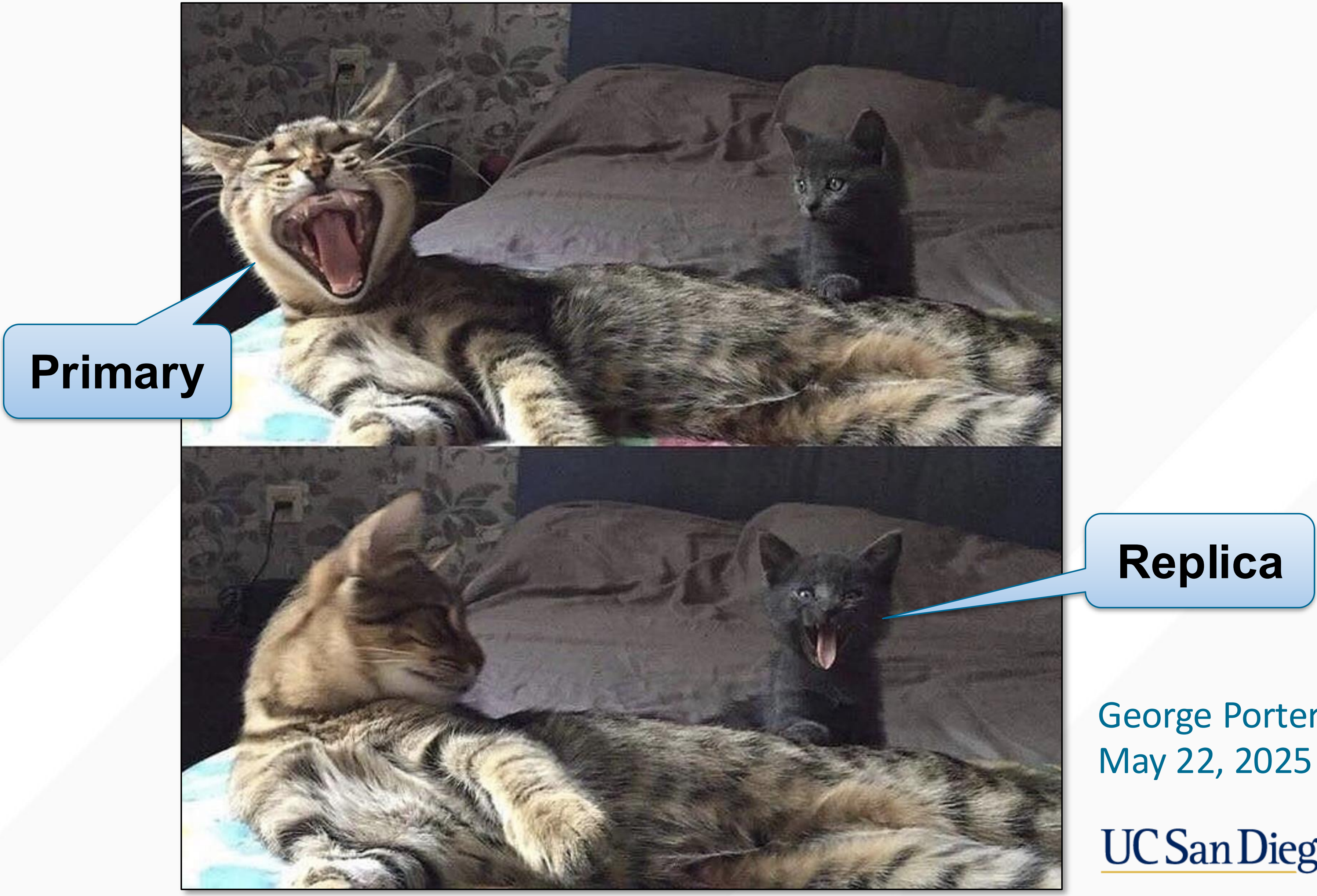
FAULT TOLERANCE VIA REPLICATION



George Porter
May 22, 2025

UC San Diego

ATTRIBUTION

- These slides incorporate material from:
    - Tanenbaum and Van Steen, Dist. Systems: Principles and Paradigms
    - Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)

UC San Diego

REQUIRED READING FOR NEXT WEEK

In Search of an Understandable Consensus Algorithm (Extended Version) by Diego Ongaro and John Ousterhout (https://raft.github.io/raft.pdf).

- Section 1, 2, 5, 8, and 11 are required reading
- Sections 3, 4, 6, 7, 9, 10, and 12 are optional and will not be covered in class
- In particular we will not be covering log compaction or membership changes!

To study for this topic, please refer to the paper. Consensus protocols are very subtle and studying these slides and/or rewatching the lecture will ***NOT*** be sufficient for obtaining a deep understanding of the RAFT protocol.

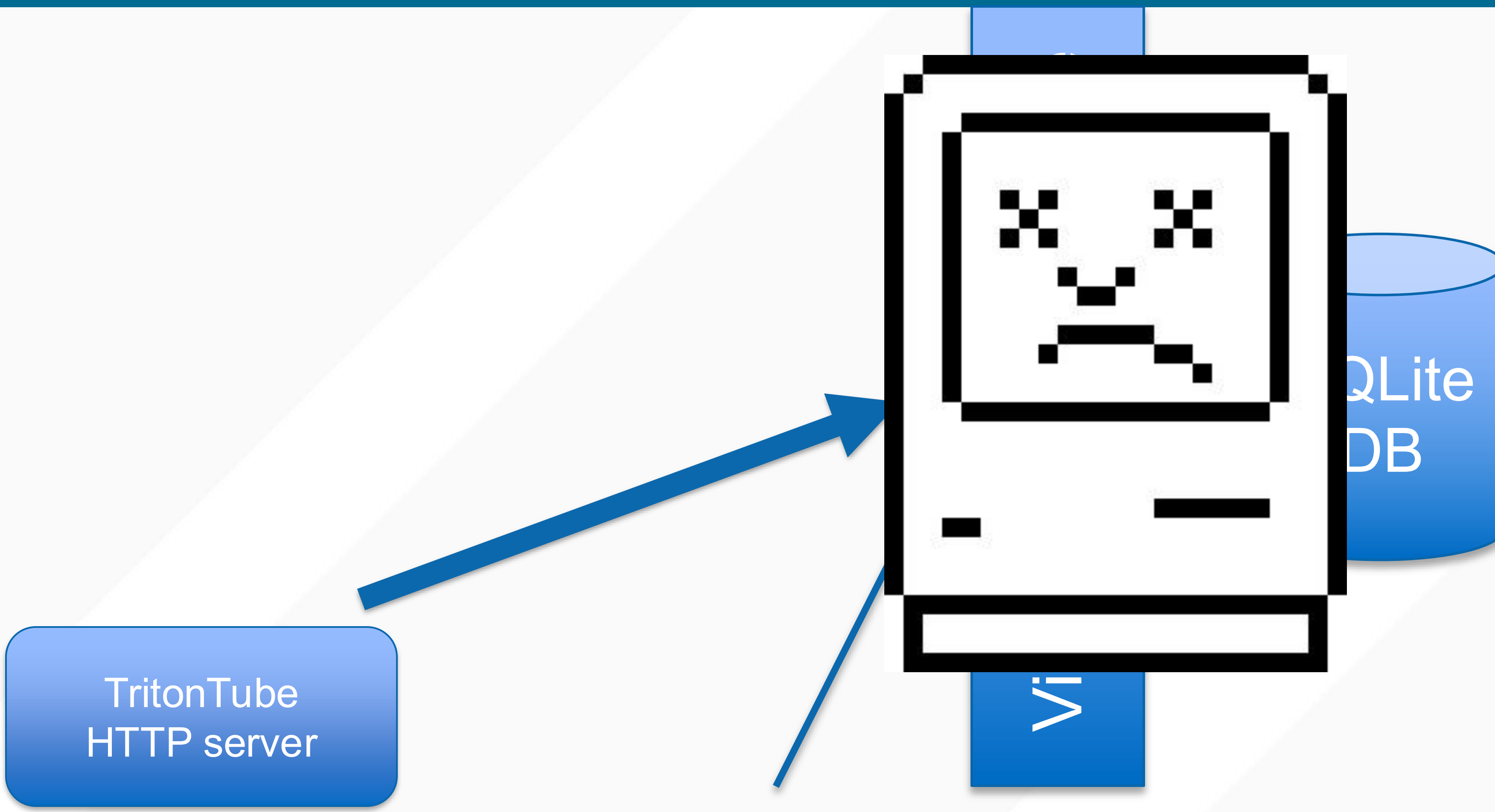
For lab 9, you will not directly implement RAFT, but you will instead incorporate the etcd system into your design (and do some experiments)

UC San Diego

Outline

1. Two-phase commit
2. Two-phase commit failure scenarios

WHAT HAPPENS IF THE METADATA STORE CRASHES?



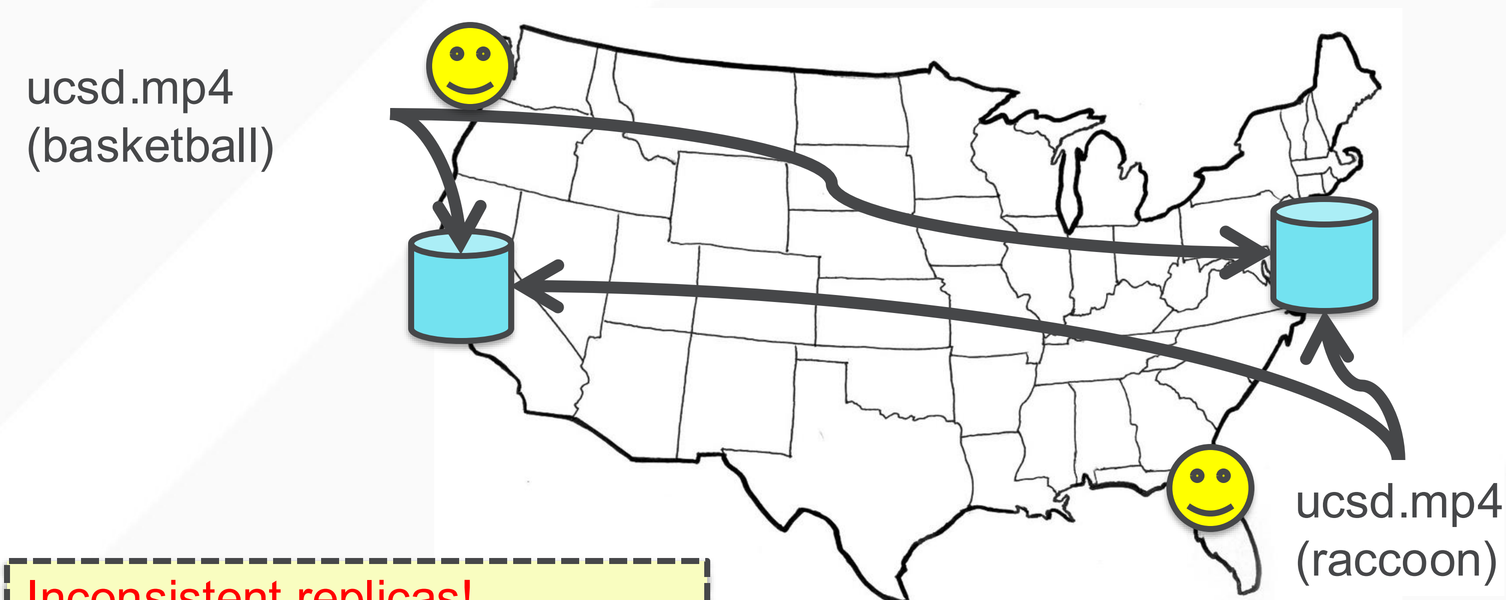
All data is lost!

MOTIVATION: MULTI-SITE METADATA REPLICATION

- TritonTube needs to be **resilient** to **whole-site failures**
- **Replicate** the metadata, keep one copy in San Francisco, one in New York

MOTIVATION: MULTI-SITE DATABASE REPLICATION

- **Replicate** the database, keep one copy in SF, one in NYC
  - Client in Seattle uploads “ucsd.mp4” (clips from March Madness basketball game)
  - Client in Florida uploads “ucsd.mp4” (video of a raccoon in front of Geisel library)



Inconsistent replicas!
Updates should have been performed in the same order at each copy

ANOTHER EXAMPLE: 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?

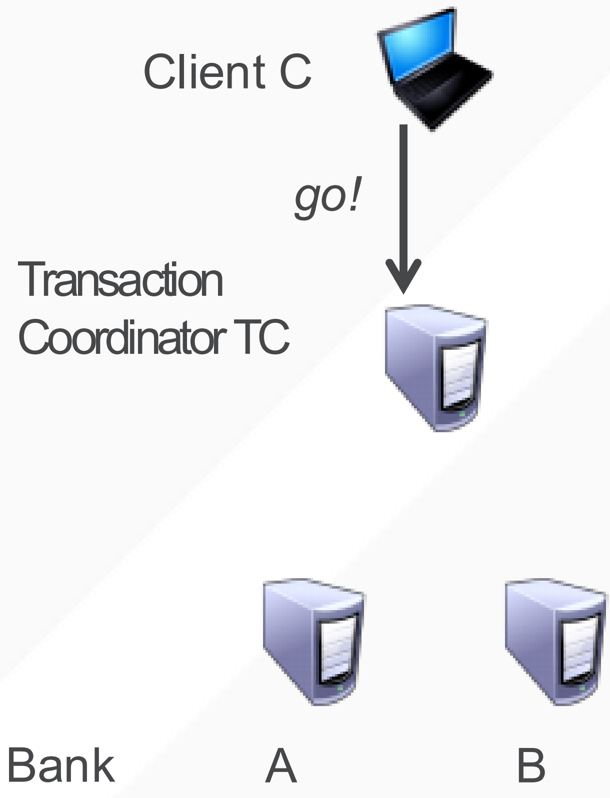
- Partition databases across multiple machines for scalability (A and B might not share a server)
- A transaction might touch more than one partition
- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?

TWO-PHASE COMMIT (2PC)

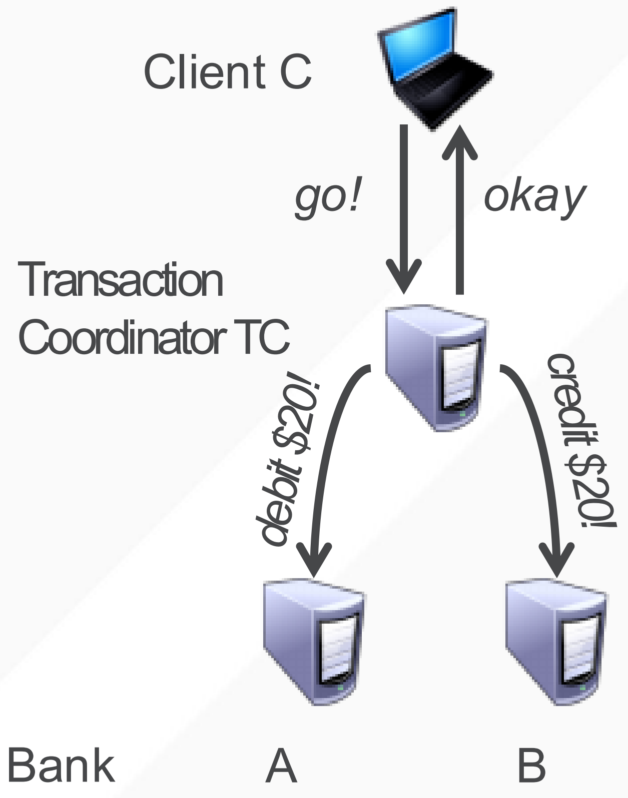
- **Goal:** General purpose, distributed agreement on some action, with failures
  - Different entities play different roles in the action
- **Running example:** Transfer money from A to B
  - Debit at A, credit at B, tell the client "okay"
  - Require **both** banks to do it, or **neither**
  - Require that **one bank never act alone**

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?

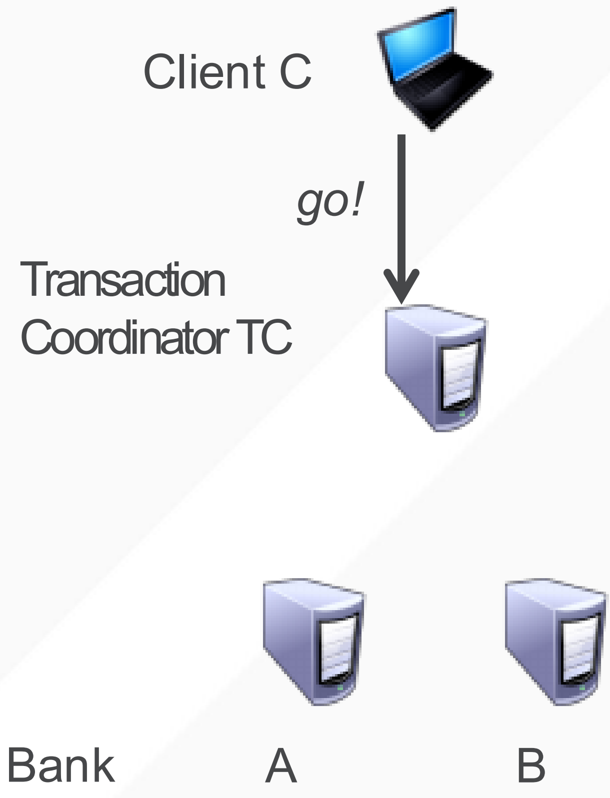
1. Not enough money in **A's** bank account?
2. **B's** bank account no longer exists?
3. **A** or **B** **crashes** before receiving message?
4. The best-effort network to **B** **fails**?
5. **TC** **crashes** after it sends *debit* to **A** but before sending to **B**?

SAFETY VERSUS LIVENESS

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

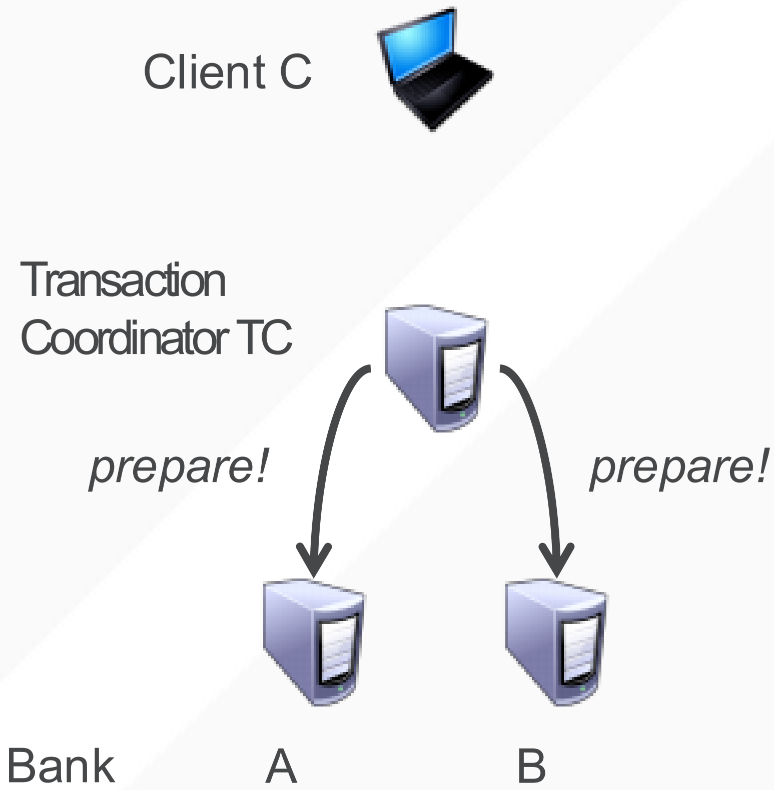
1. Safety
   - If **one commits, no one aborts**
   - If **one aborts, no one commits**
2. Liveness
   - If **no failures** and **A** and **B** can commit, **action commits**
   - If **failures,** reach a conclusion ASAP

A *CORRECT* ATOMIC COMMIT PROTOCOL



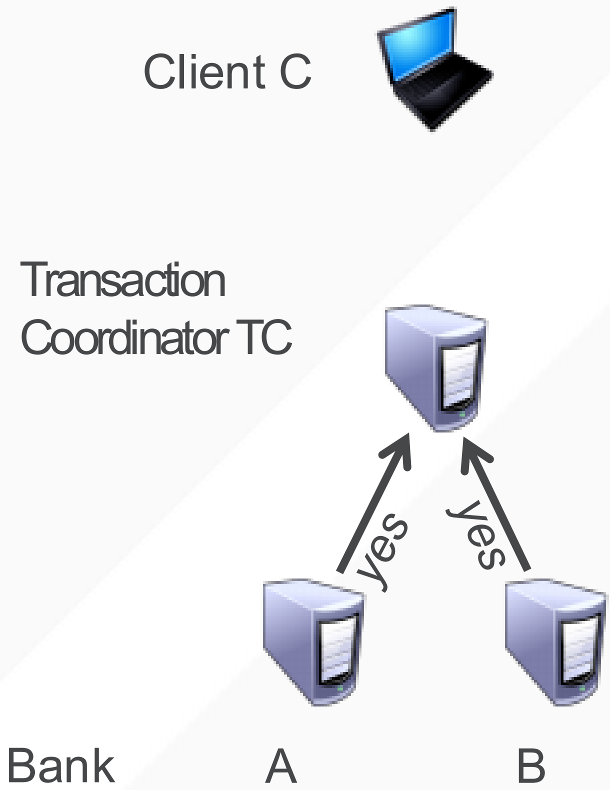
1. C → TC: *"go!"*

A *CORRECT* ATOMIC COMMIT PROTOCOL



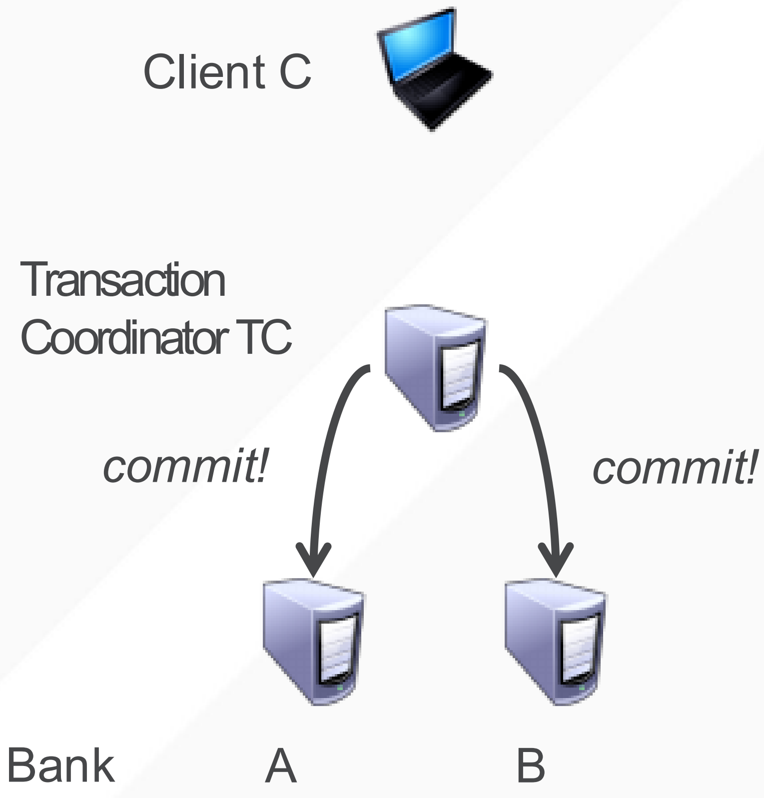
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*

A *CORRECT* ATOMIC COMMIT PROTOCOL



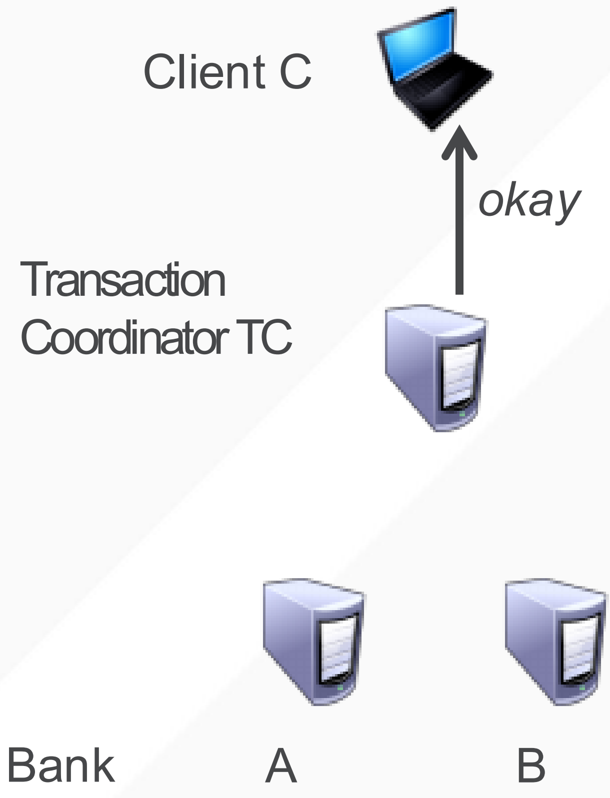
1. **C → TC:** *"go!"*
2. **TC → A, B:** *"prepare!"*
3. **A, B → P:** *"yes"* or *"no"*

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

DURABILITY ACROSS REBOOTS

```
FSYNC(2)                  Linux Programmer's Manual                  FSYNC(2)

NAME    top

       fsync, fdatasync - synchronize a file's in-core state with storage
       device

SYNOPSIS    top

       #include <unistd.h>

       int fsync(int fd);

       int fdatasync(int fd);

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       fsync():
           Glibc 2.16 and later:
               No feature test macros need be defined
           Glibc up to and including 2.15:
               _BSD_SOURCE || _XOPEN_SOURCE
                   || /* since glibc 2.8: */ _POSIX_C_SOURCE >= 200112L
       fdatasync():
           _POSIX_C_SOURCE >= 199309L || _XOPEN_SOURCE >= 500
```

GO'S OS PACKAGE: OS.FILE.SYNC()

```
type File
    func Create(name string) (*File, error)
    func CreateTemp(dir, pattern string) (*File, error)
    func NewFile(fd uintptr, name string) *File
    func Open(name string) (*File, error)
    func OpenFile(name string, flag int, perm FileMode) (*File, error)
    func (f *File) Chdir() error
    func (f *File) Chmod(mode FileMode) error
    func (f *File) Chown(uid, gid int) error
    func (f *File) Close() error
    func (f *File) Fd() uintptr
    func (f *File) Name() string
    func (f *File) Read(b []byte) (n int, err error)
    func (f *File) ReadAt(b []byte, off int64) (n int, err error)
    func (f *File) ReadDir(n int) ([]DirEntry, error)
    func (f *File) ReadFrom(r io.Reader) (n int64, err error)
    func (f *File) Readdir(n int) ([]FileInfo, error)
    func (f *File) Readdirnames(n int) (names []string, err err
    func (f *File) Seek(offset int64, whence int) (ret int64, err
    func (f *File) SetDeadline(t time.Time) error
    func (f *File) SetReadDeadline(t time.Time) error
    func (f *File) SetWriteDeadline(t time.Time) error
    func (f *File) Stat() (FileInfo, error)
    func (f *File) Sync() error
    func (f *File) SyscallConn() (syscall.RawConn, error)
    func (f *File) Truncate(size int64) error
    func (f *File) Write(b []byte) (n int, err error)
    func (f *File) WriteAt(b []byte, off int64) (n int, err error)
    func (f *File) WriteString(s string) (n int, err error)
type FileInfo
```

func (*File) Sync

```
func (f *File) Sync() error
```

Sync commits the current contents of the file to stable storage. Typically, this means flushing the file system's in-memory copy of recently written data 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**

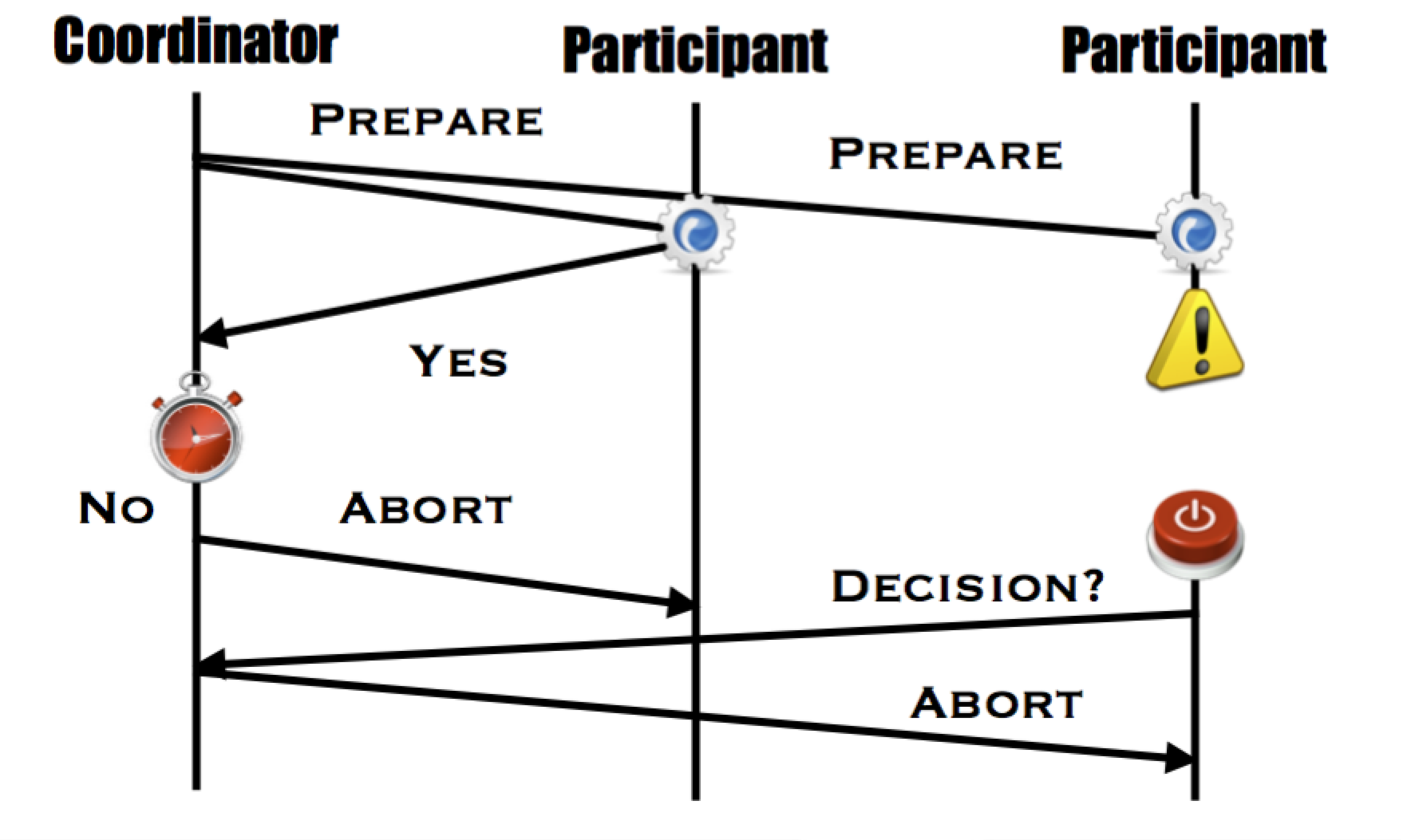
ROADMAP

- Today: Two-phase commit (2PC)
- Next week:
  - Replicated state machines
  - How the transaction coordinator can use 2PC to durably send updates to multiple replicated state machines
  - How to “elect” a node to serve as the transaction coordinator
- Net result:
  - The above topics form the basis of the “RAFT” consensus protocol (used in many systems, etcd, lab 9, etc)

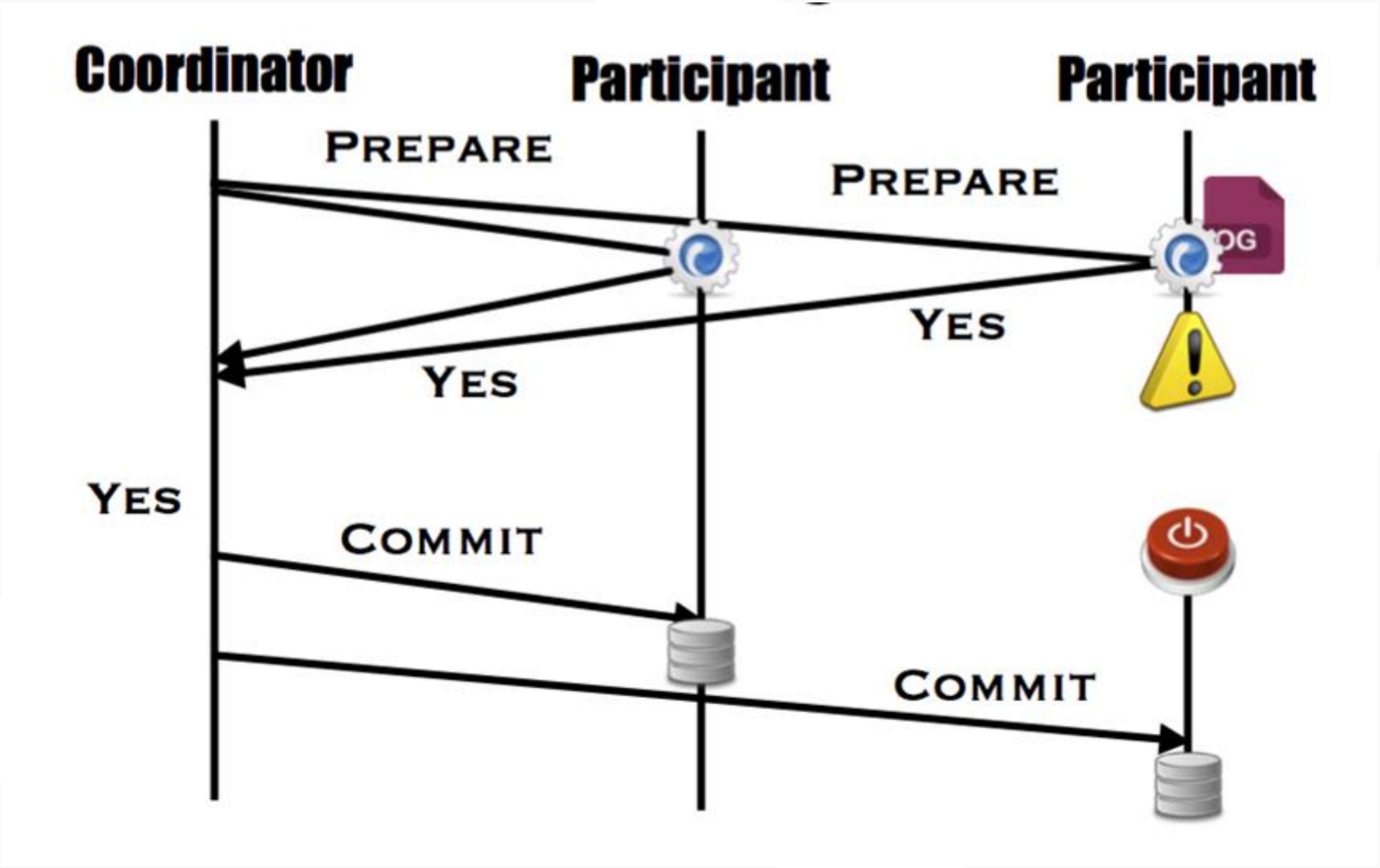
Outline

1. Two-phase commit
2. Two-phase commit failure scenarios

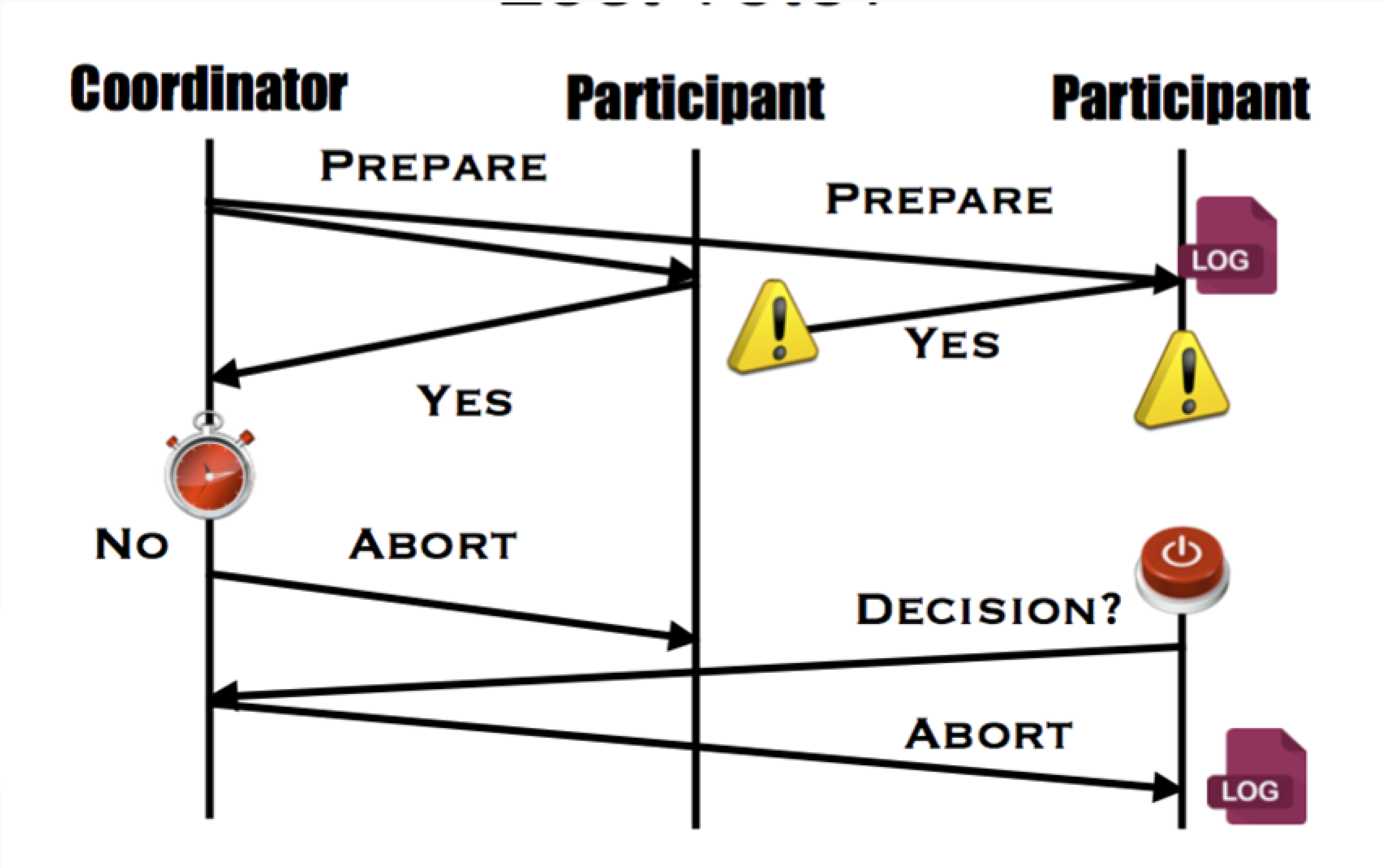
WHAT IF PARTICIPANT FAILS BEFORE SENDING RESPONSE?



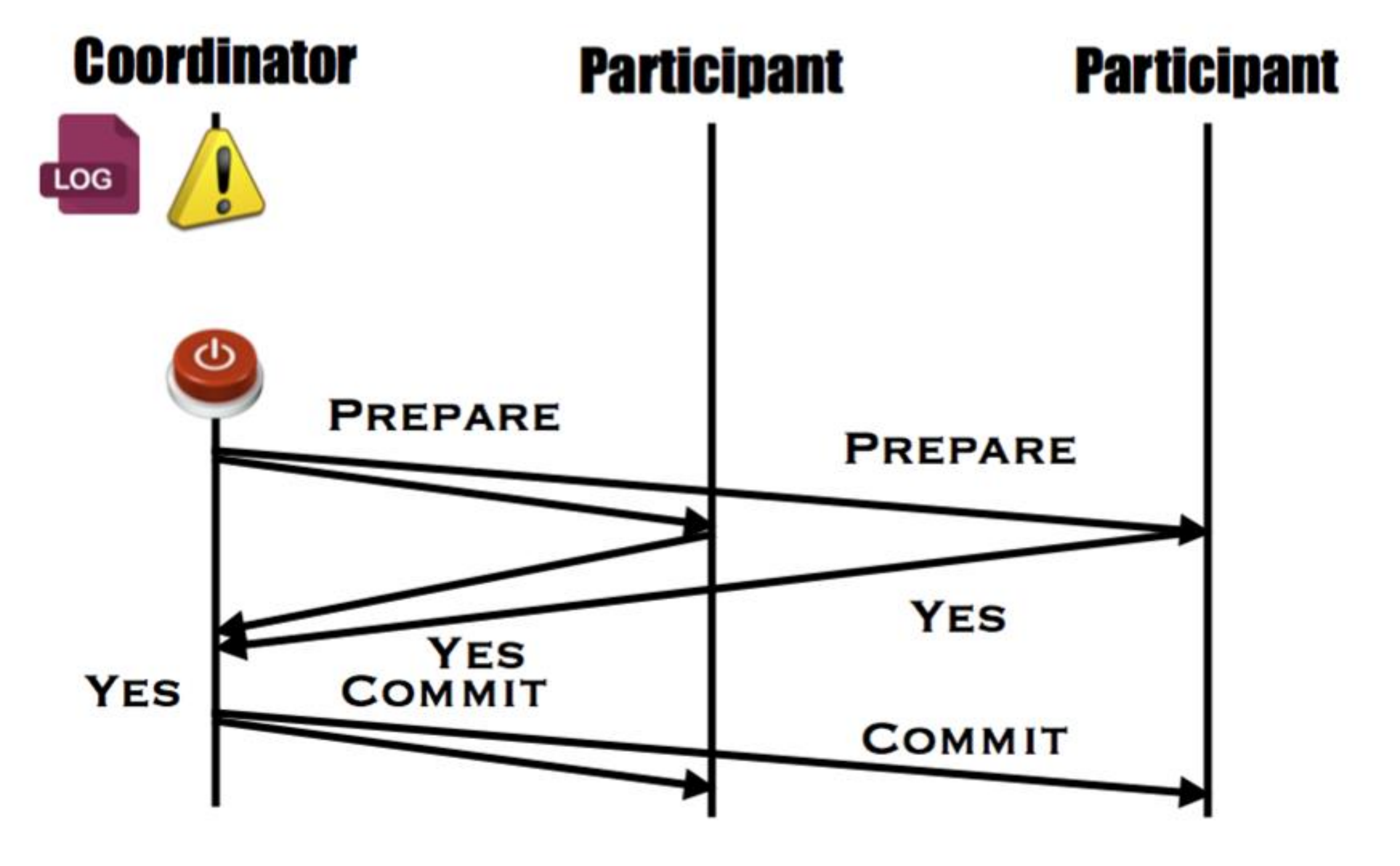
WHAT IF PARTICIPANT FAILS AFTER SENDING VOTE



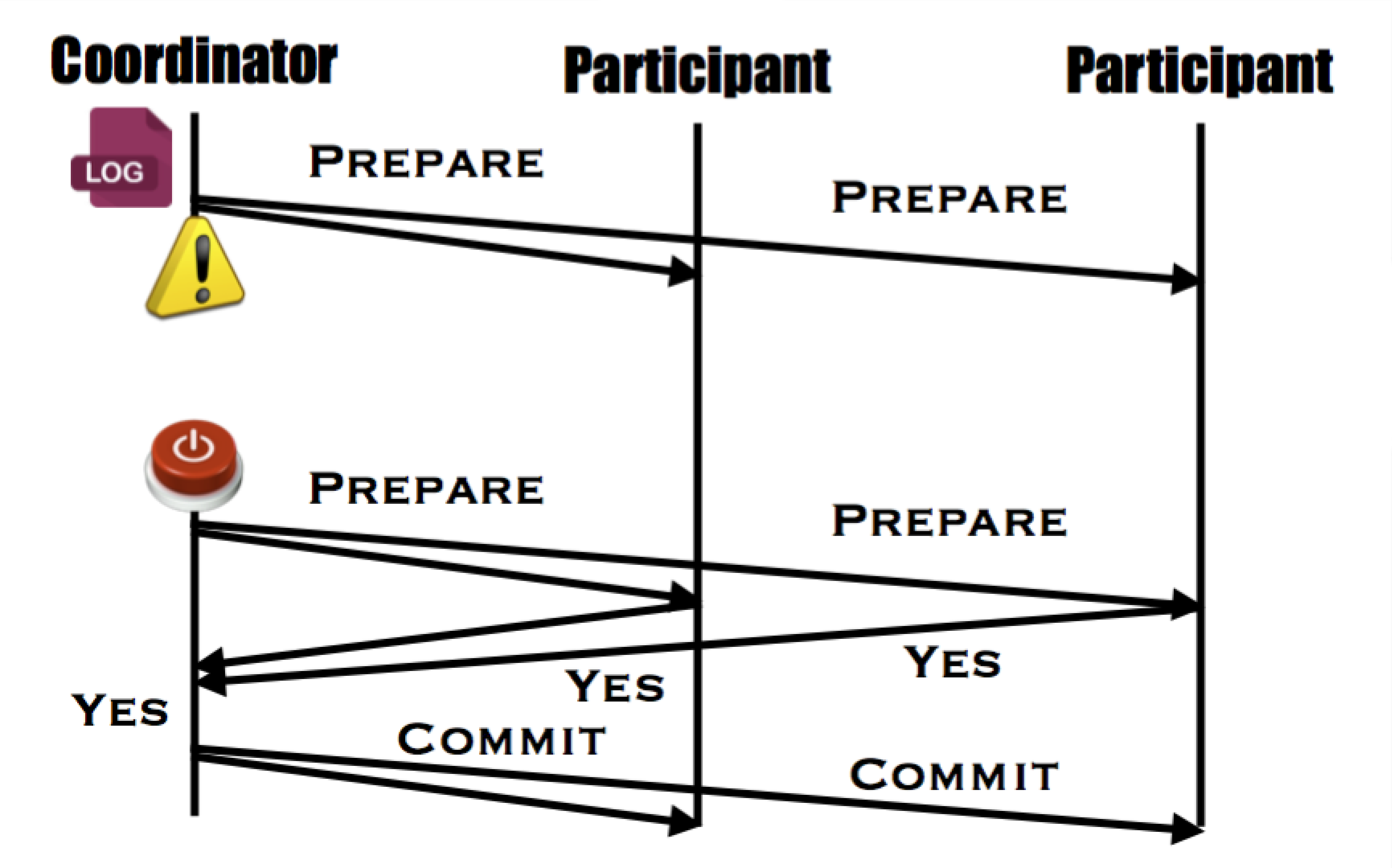
WHAT IF PARTICIPANT LOST A VOTE?



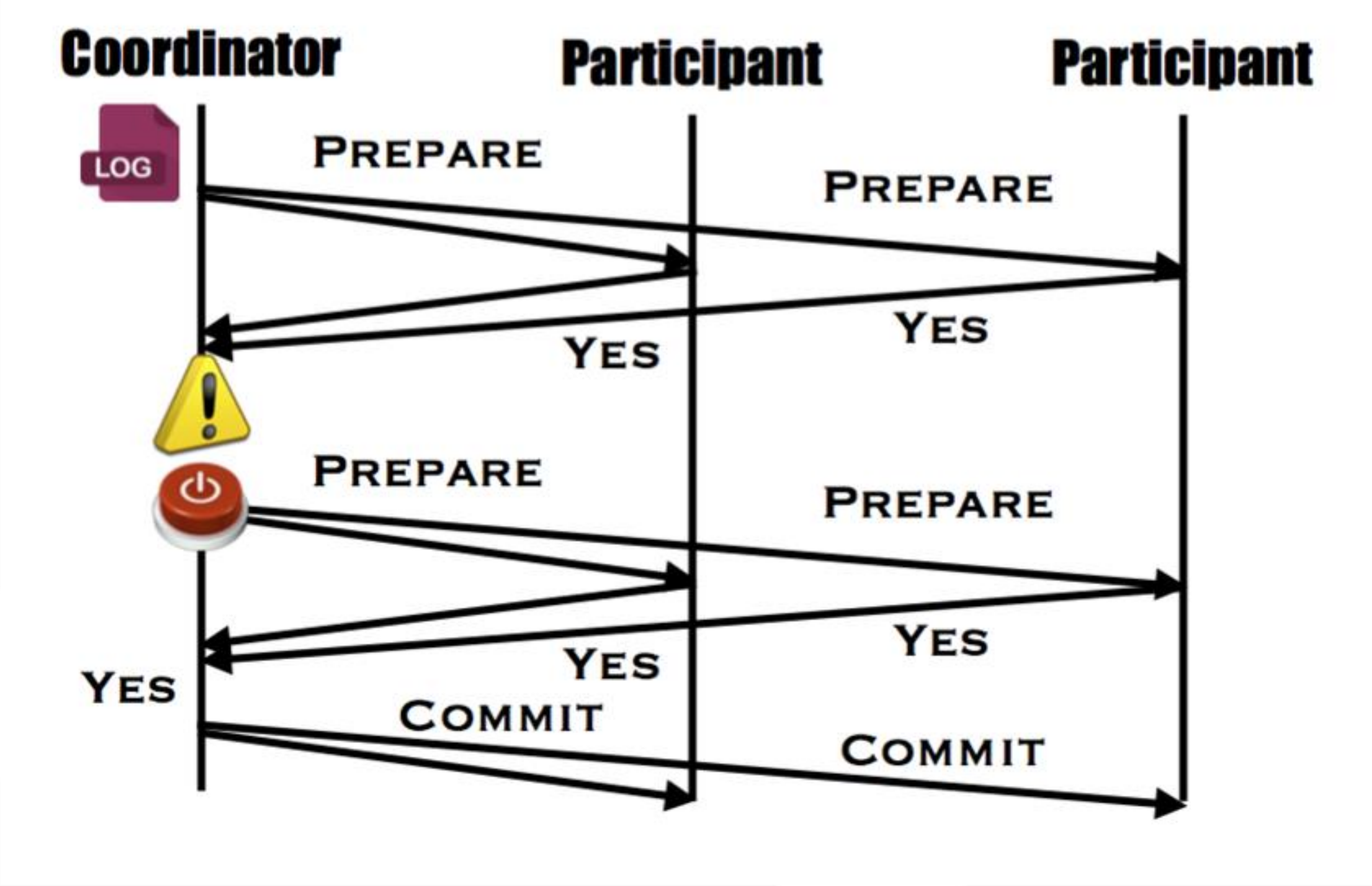
WHAT IF COORDINATOR FAILS BEFORE SENDING PREPARE?



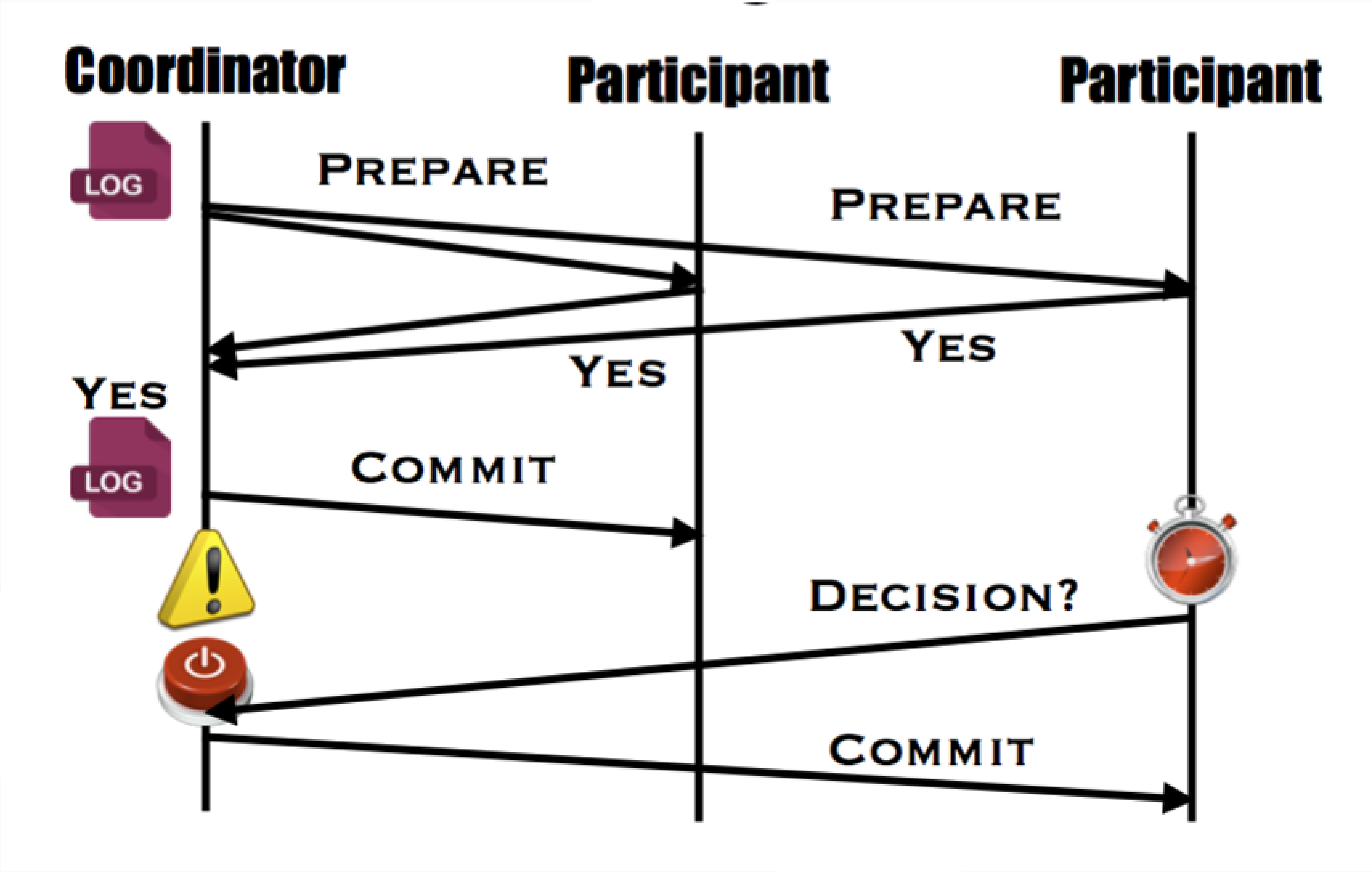
WHAT IF COORDINATOR FAILS AFTER SENDING PREPARE?



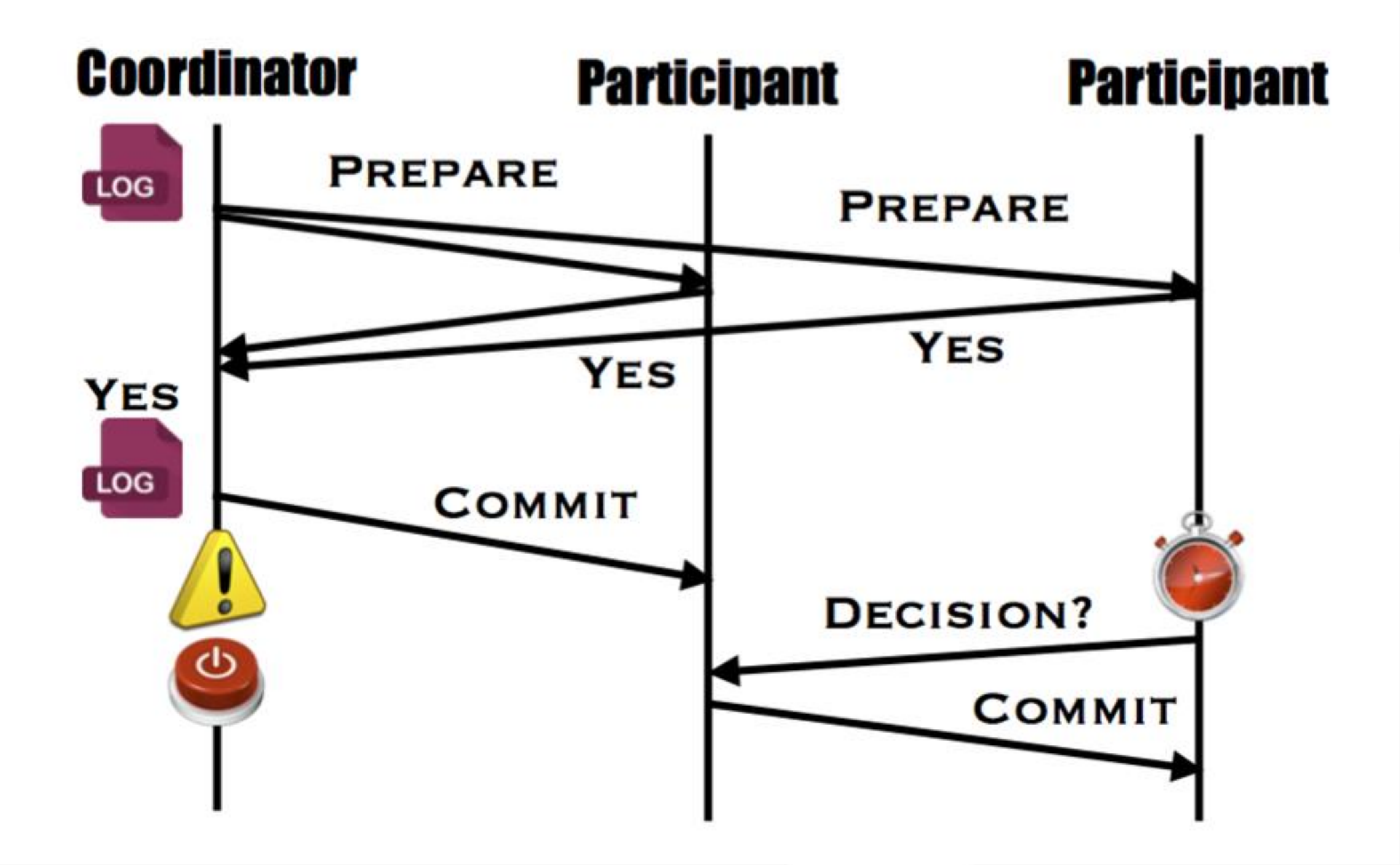
WHAT IF COORDINATOR FAILS AFTER RECEIVING VOTES



WHAT IF COORDINATOR FAILS AFTER SENDING DECISION?



DO WE NEED THE COORDINATOR?



UC San Diego