UNIVERSITY of VIRGINIA

CS4740
CLOUD COMPUTING

Two-Phase Commit

Prof. Chang Lou, UVA CS, Fall 2025

Amazon's AWS strikes AI cloud partnership with NBA

By **Zaheer Kachwala**

October 1, 2025 7:35 AM EDT · Updated October 1, 2025

Oct 1 (Reuters) - Amazon.com's (AMZN.O) cloud unit, Amazon Web Services, announced a multi-year partnership with the National Basketball Association to launch new AI-powered features and data insights from games, the organizations said on Wednesday.

The deal, which did not have a financial value attached, will introduce "NBA Inside the Game", a basketball platform that will turn data points into insights and interactive experiences.

> The Reuters Tariff Watch newsletter is your daily guide to the latest global trade and tariff news. Sign up here.

Sports firms have increasingly adopted artificial intelligence, striking partnerships with cloud providers to use machine learning to provide in-depth statistics for fans.

The "NBA Inside the Game" platform will offer a suite of features for live broadcasts and across the NBA App, website and other social channels, they said.

The new AI features and data will also be available to teams for review.

"Whenever we build a statistic, there is always input from our teams, because we want to make sure that these services we build not only are great for our fans, but also help our teams improve their strategy on court," NBA's head of media operations and technology, Ken DeGennaro told Reuters.

The platform will track data from players to generate AI statistics such as defensive positions of teams, data on shot success percentage and strategy, he added.

Earlier this year, Microsoft (MSFT.O) signed a five-year partnership with the English Premier League under which the cloud giant will infuse its AI copilot into the league's digital platforms.

Explosive battery blaze in South Korea ‘paralyzes’ vital government services

SEP 27, 2025

By Laura Sharman

South Korean Prime Minister Kim Min-seok speaks during an anti-terrorism drill at the Korea Multi-purpose Accelerator Complex in Gyeongju, South Korea, in September 2025. *(YONHAP/EPA/Shutterstock)*

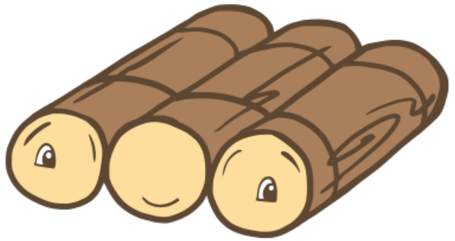
A major fire at South Korea’s national data center has halted hundreds of online government services including postal and tax facilities, Reuters reported.

The blaze likely began with an explosion of a battery at the National Information Resources Service in Daejeon city, about 85 miles (130 kilometers) south of the capital Seoul, at around 8:20 p.m. local time on Friday, the news agency reported.

That led to a “thermal runaway” releasing extreme heat in the server room, making it difficult for firefighters to contain the inferno, officials told the agency.

Prime Minister Kim Min-seok said the fire had “paralyzed” the government’s internal digital platform, shutting down its official email system and several websites.

LAB 2A IS RELEASED



- **Lab2A: Raft - Leader Election**
  - Deadline: 10/20 23:59:59 EST
  - A lot of interesting questions to answer in your lab 2A implementation
    - How can a group of votes decide the unique leader? What if the old leader dies? What if some network failures happen? How to send RPCs to multiple nodes in parallel?...
- **My advice (*again*): Start the lab 2A immediately**
  - Don't wait for lectures: they only cover high-level designs
  - You'll learn much more from paper and hands-on experience
  - **Refer to the overview session recordings on Canvas!**
  - In summary: start lab 2 **today**!

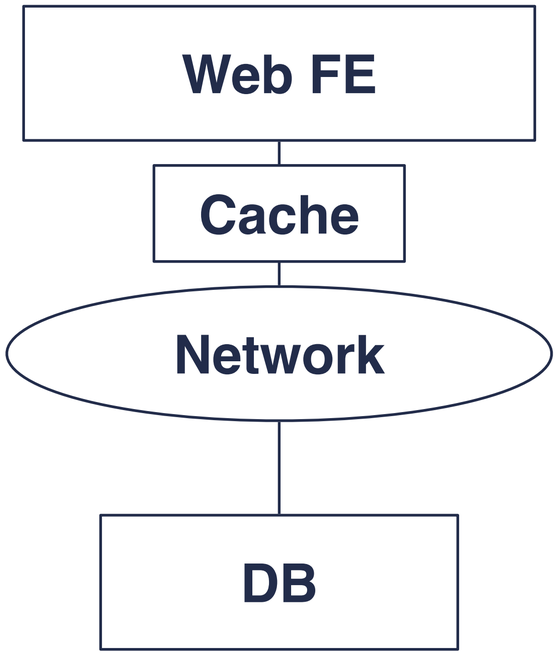
AGREEMENT

- We discussed two types of agreements
  - Atomic commitment: e.g., when to meet
  - Consensus: e.g., which zoom link to use
- Why we need such agreements in DS?
  - Atomic commitment <= Scalability: sharding
  - Consensus <= Fault tolerance: replication
- Today we discuss a solution 2PC for atomic commitment
  - We have discussed transactions in single-node settings
  - Let's discuss how distributed transactions are implemented

OUTLINE

- Context
- Two-phase commit (2PC)
- 2PC failure scenarios
- 2PC limitations

EXAMPLE: WEB SERVICE ARCHITECTURE

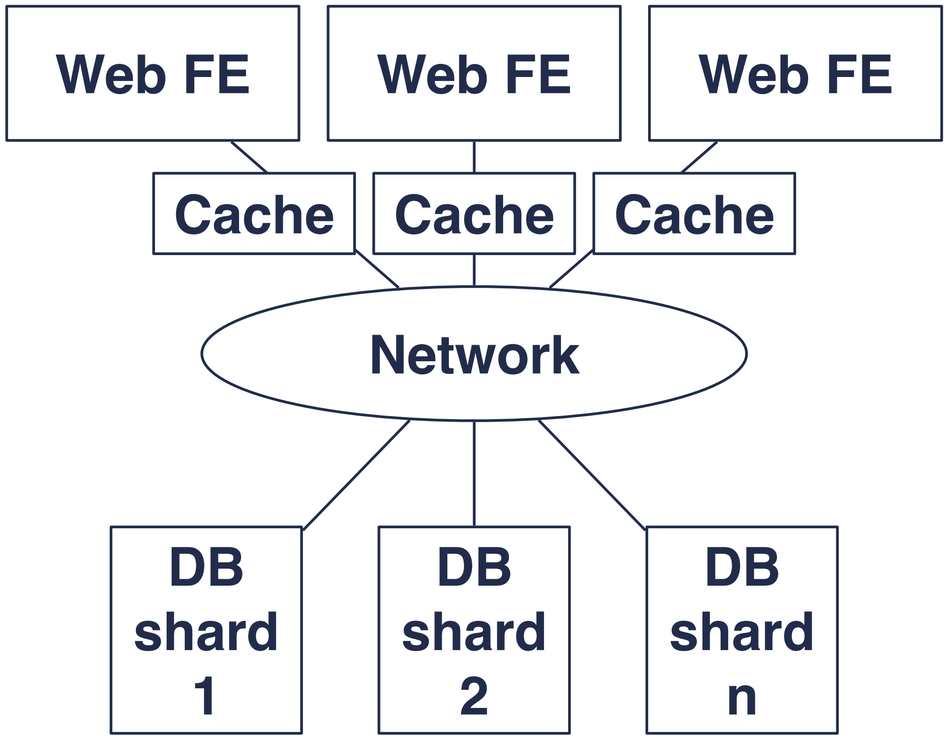


- Web front end (FE), database server (DB), network. FE is stateless, all state in DB.
- Suppose the FE implements a banking application (supporting account transfers, listings, and other functionality).
- Suppose the DB supports ACID transactions and the FE uses transactions.

Question: How do we make this:

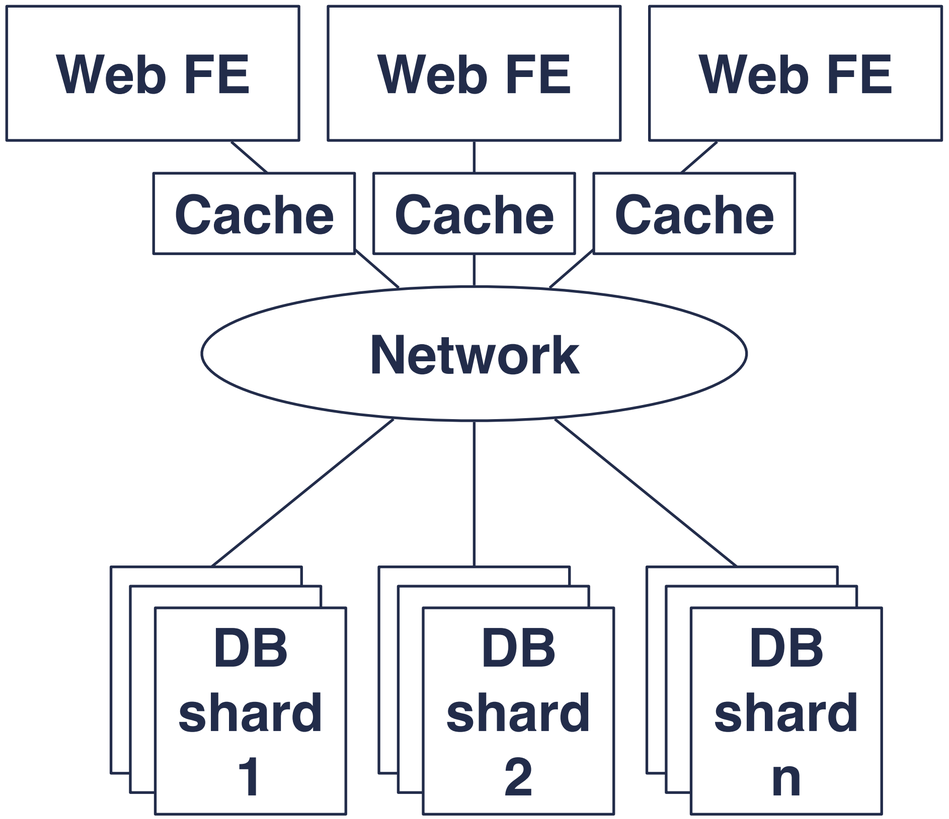
- scalable?
- fault tolerant?

SCALABILITY: SHARDING



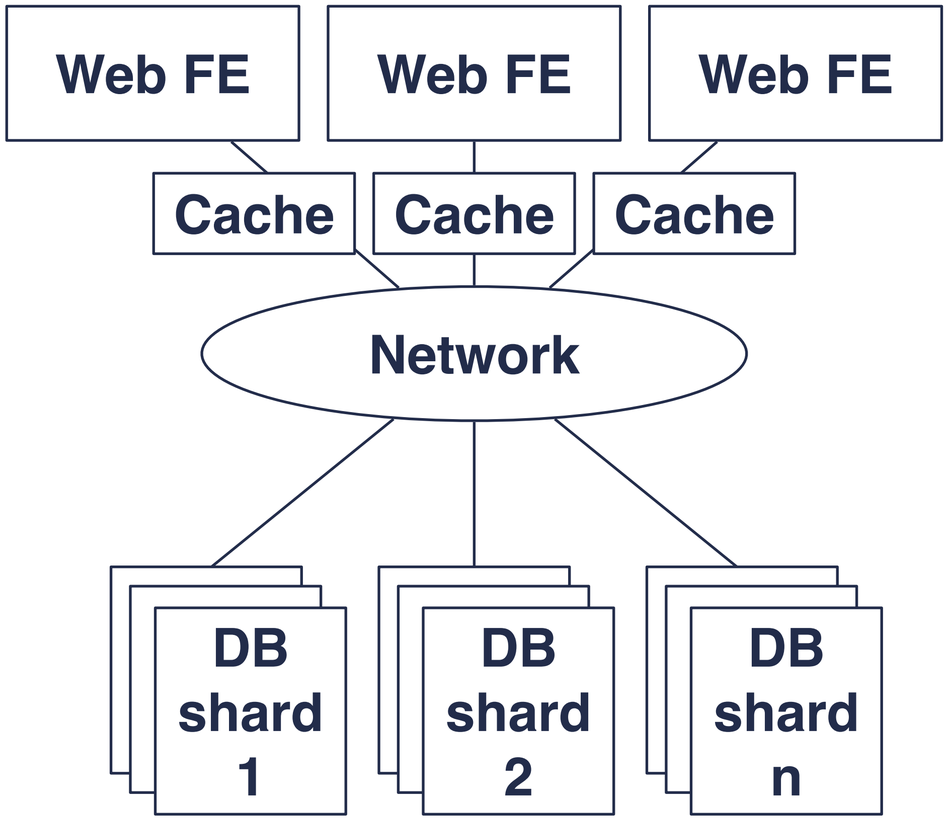
- FE and DB are both sharded:
  - FEs accept requests from end-users' browsers and process them concurrently.
  - DB is sharded, say by user IDs.
- Suppose each DB backend is on its own transactional (ACID). Then, FE issues transactions against one or more DB shards.

FAULT TOLERANCE: REPLICATION



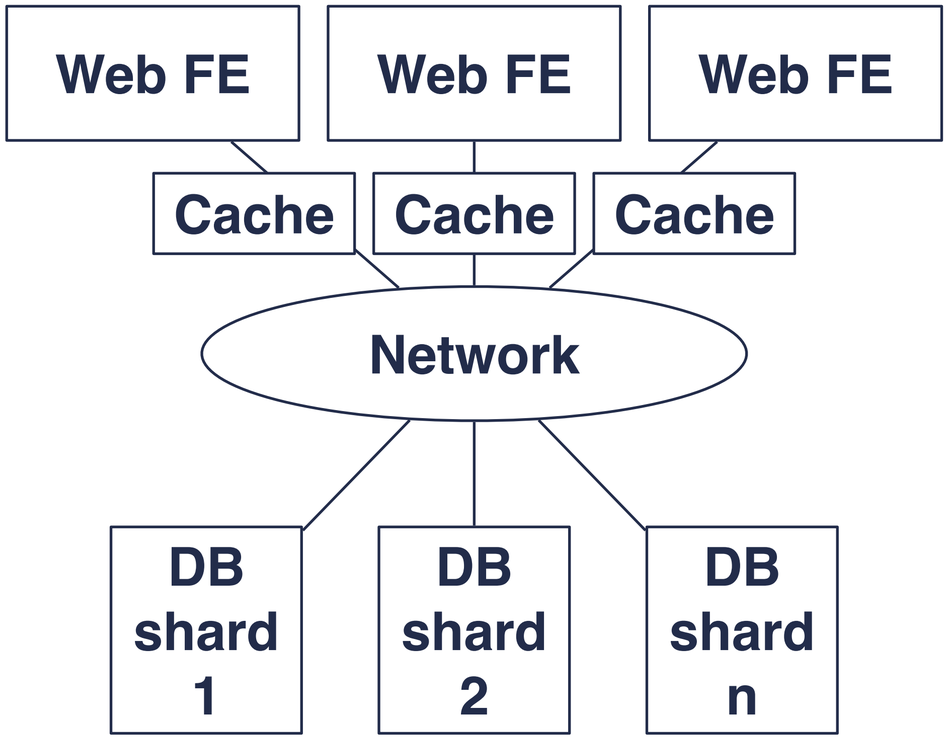
- FE is stateless, so the fact that it is shared means it's also replicated/fault tolerant.
- But DB is stateful, so active replication is needed for each shard. Each shard is managed by a replica group, which cooperate to keep themselves up to date with respect to the updates.
- FE sends requests for DB different shards go to different replica groups.

CHALLENGES



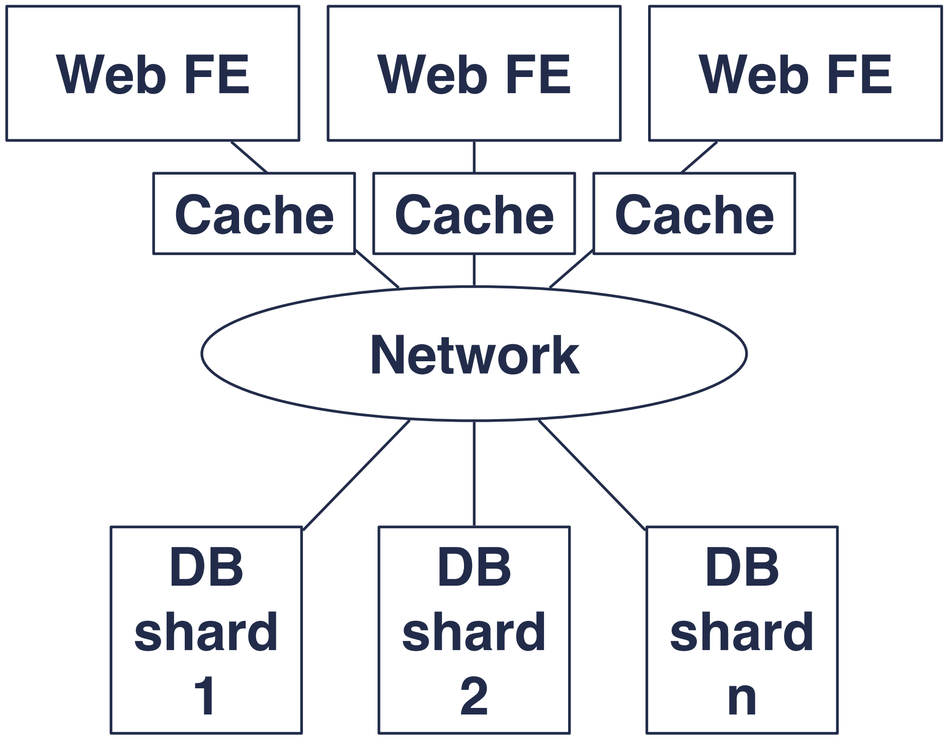
— Question: What are the challenges of implementing ACID across the entire sharded & replicated, DB service?

CHALLENGES DUE TO SHARDING



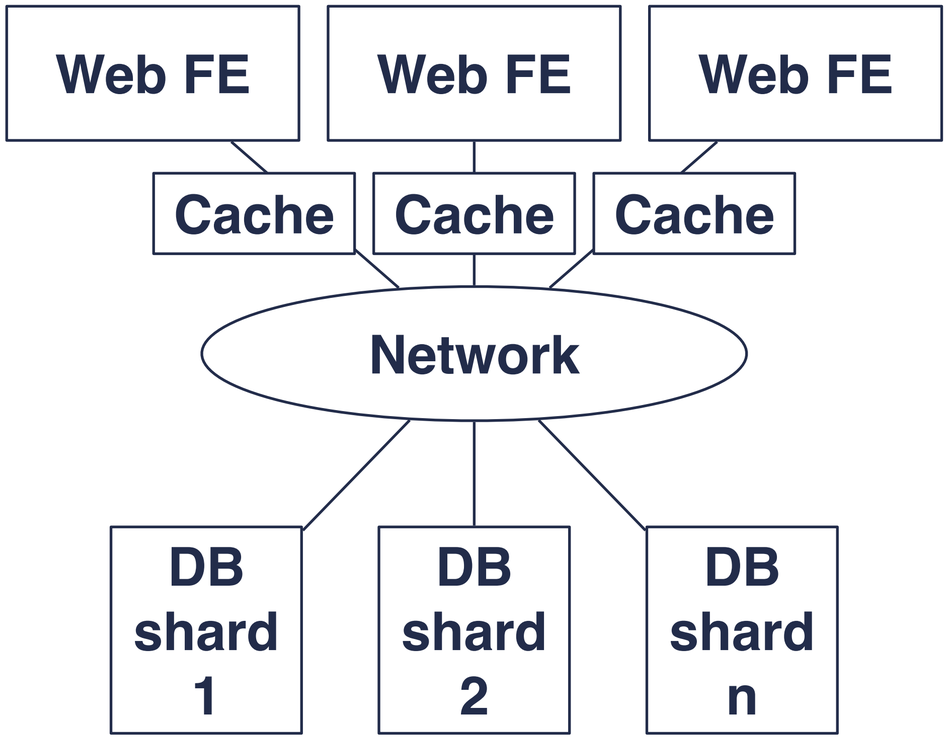
- Ignore replication. Implementing ACID across all DB shard servers:
- Case 1: No transactions ever span multiple shards. Easy: individual DB shard performs transaction.
- Case 2: Transactions can span multiple shards. Challenge: shards participating on any transaction need to agree on (1) whether or not to commit a transaction and (2) when to release the locks.

CHALLENGES DUE TO SHARDING



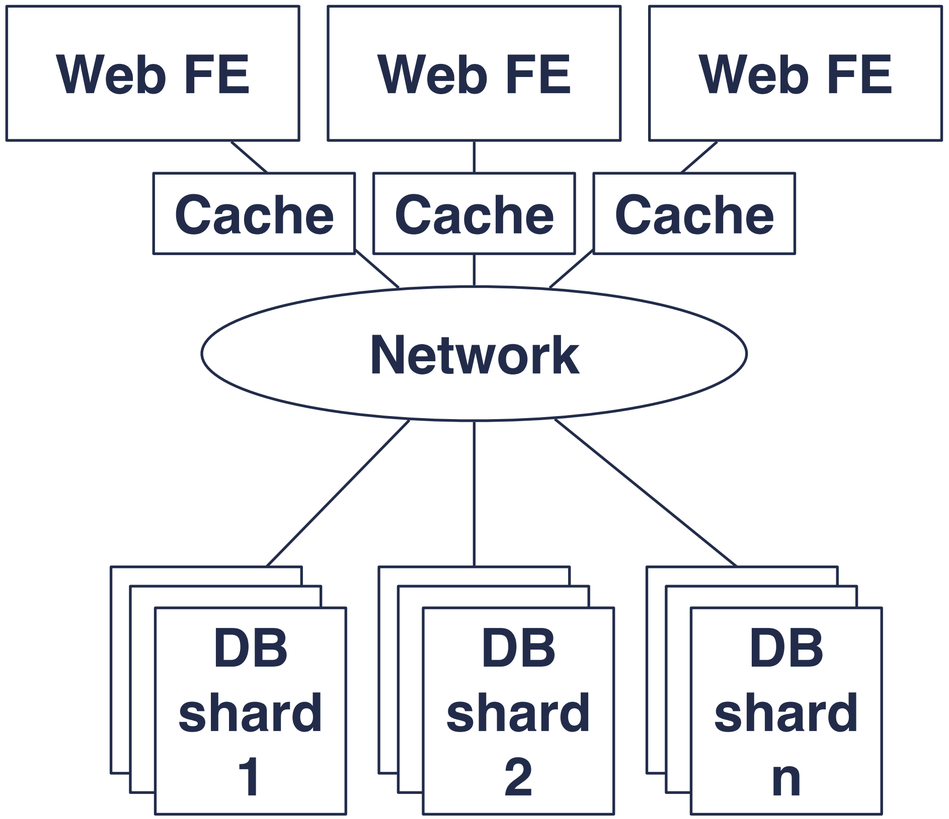
- Example:
  - Say FE service is a banking service that supports the TRANSFER and REPORT_SUM functions from the previous lecture.
  - If the two accounts are stored on different shards, then the two operations (deduct from one and add to the other) will need to be executed either both or neither.
  - Unfortunately, the two machines can fail, or decide to unilaterally abort, INDEPENDENTLY.

CHALLENGES DUE TO SHARDING



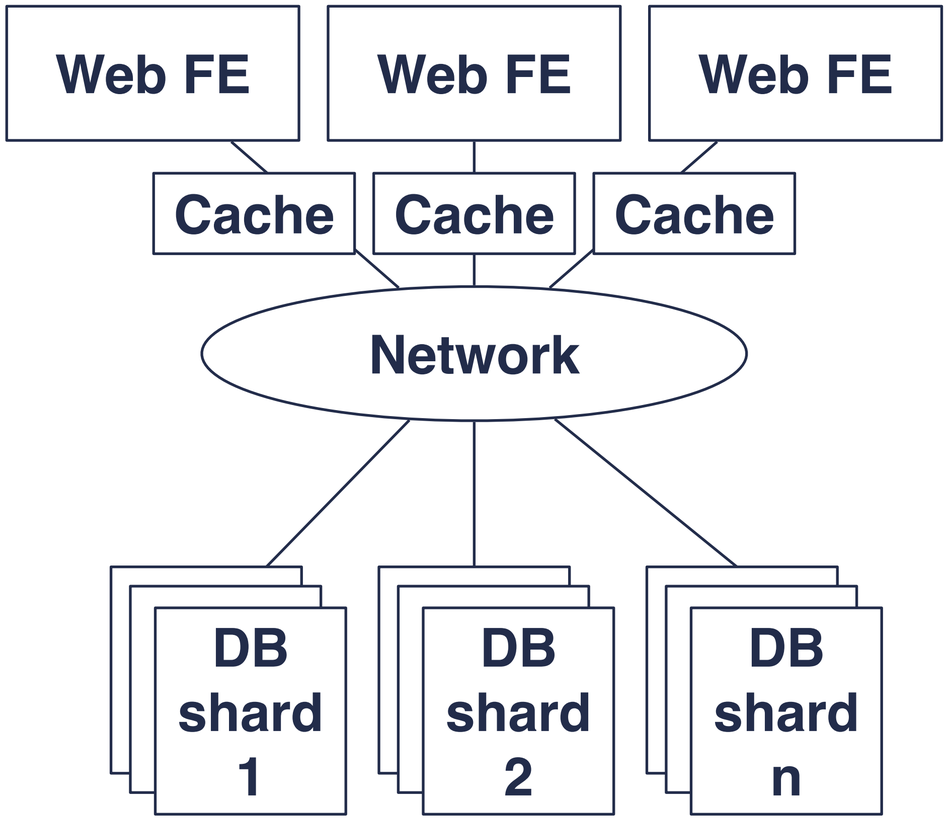
- Example:
  - So, you need an agreement protocol, and in this case the most suitable is an atomic commitment protocol (why?).
  - Well-known atomic commitment protocol: two-phase commit.

CHALLENGES DUE TO REPLICATION



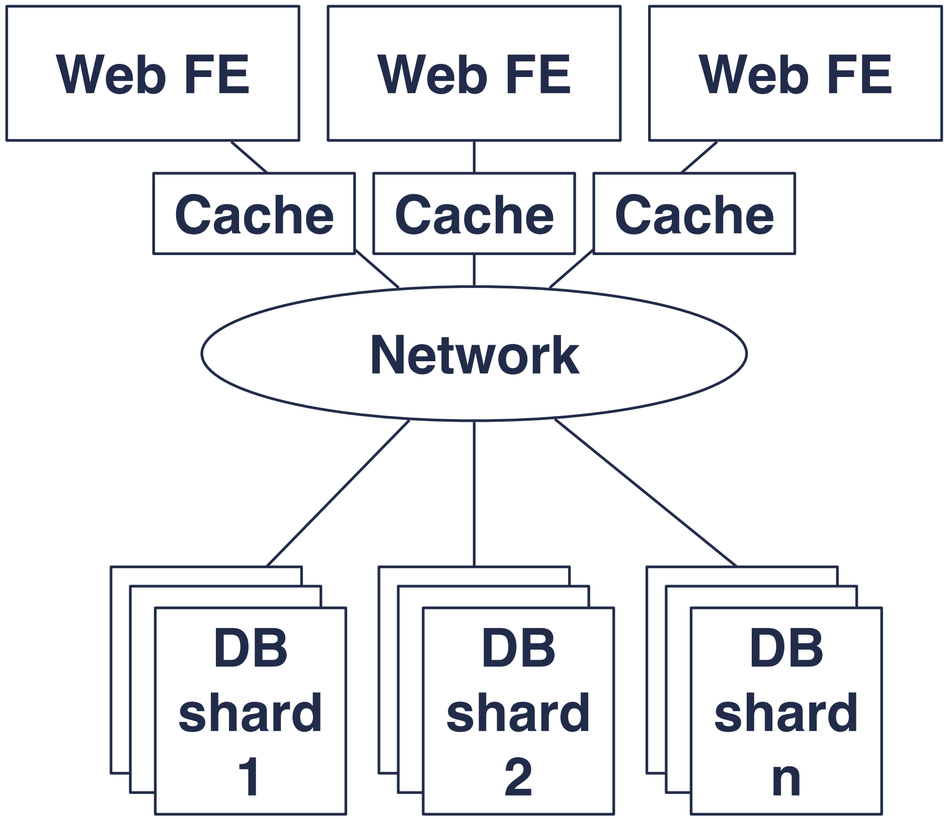
- Ignore sharding. Implementing ACID across all replicas of a given shard:
  - Challenge: All replicas of the shard must execute all operations in the same order.
  - If the operations are deterministic, then agreeing on the order of keeps the copies of the database on the different replicas will evolve identically, i.e., they will all be kept consistent.

CHALLENGES DUE TO REPLICATION



- Example
  - Suppose there are two transactions, each with a single operation, against the same cell in the database:
    - TX1: x += 1
    - TX2: x *= 2
  - Internally, all three replicas are ACID databases, so they will serialize these transactions, e.g., either (TX1, TX2) OR (TX2, TX1).
  - If Replica A processes (TX1, TX2) and Replica B processes (TX2, TX1), then after executing these transactions, the DB copies on the two replicas will diverge to x=8 and x=7, respectively.

CHALLENGES DUE TO REPLICATION



- Example
  - The problem of agreement on the order in which to execute operations can be cast as an instance of the consensus problem (why?).
  - Well known consensus protocol: Paxos, Raft
  - We study these protocols next time.

Two-Phase Commit (2PC)

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)

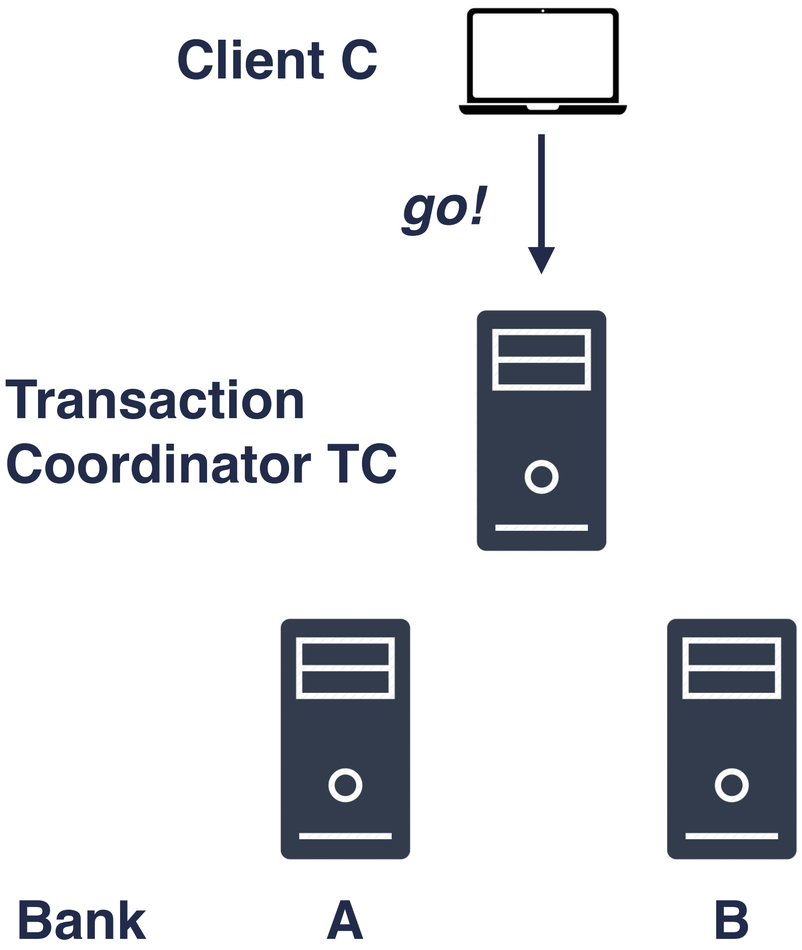
- A transaction might touch more than one partition

- How do we guarantee that all of the partitions commit the transactions or none commit the transactions?

TWO-PHASE COMMIT (2PC)

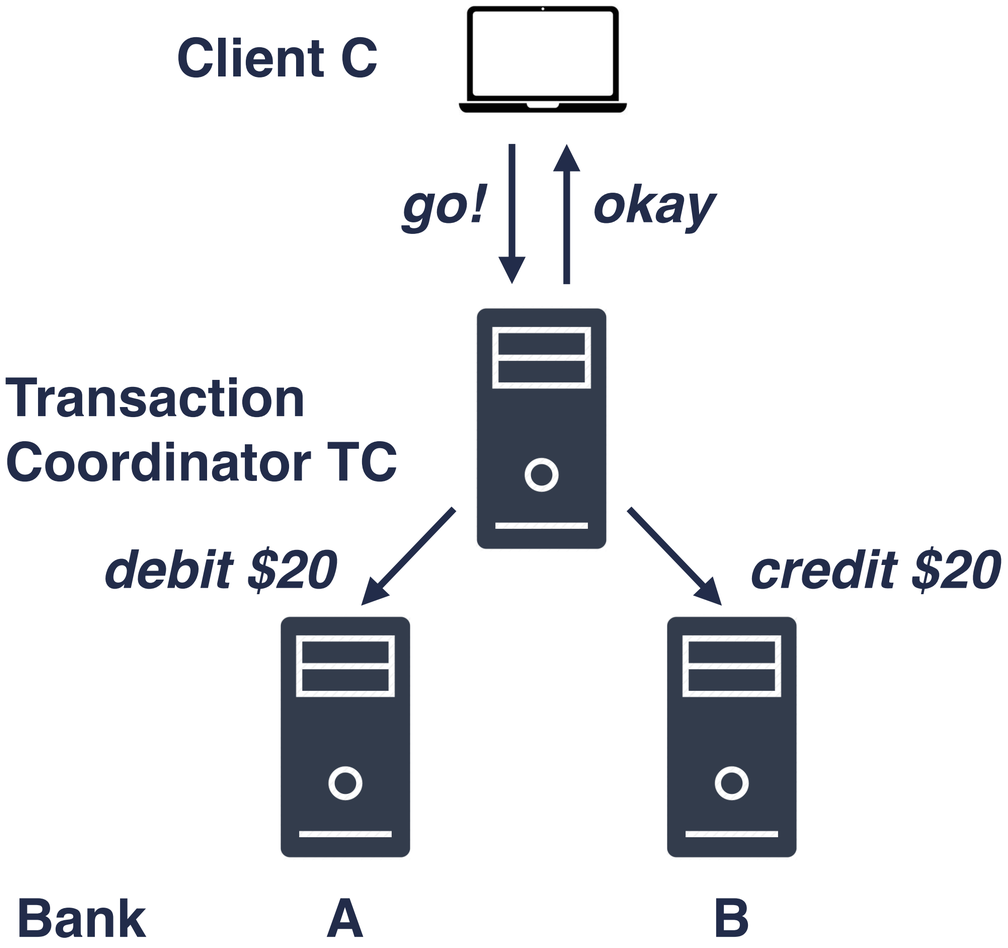
- **Goal**: General purpose, distributed agreement on some action, with failures
  - Different entities play different roles in the action
- Running **example**: Transfer money from A to B
  - Debit at A, credit at B, tell the client “okay”
  - Require both banks to do it, or neither
  - Require that one bank never act alone
- This is an **all-or-nothing** atomic commit protocol

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

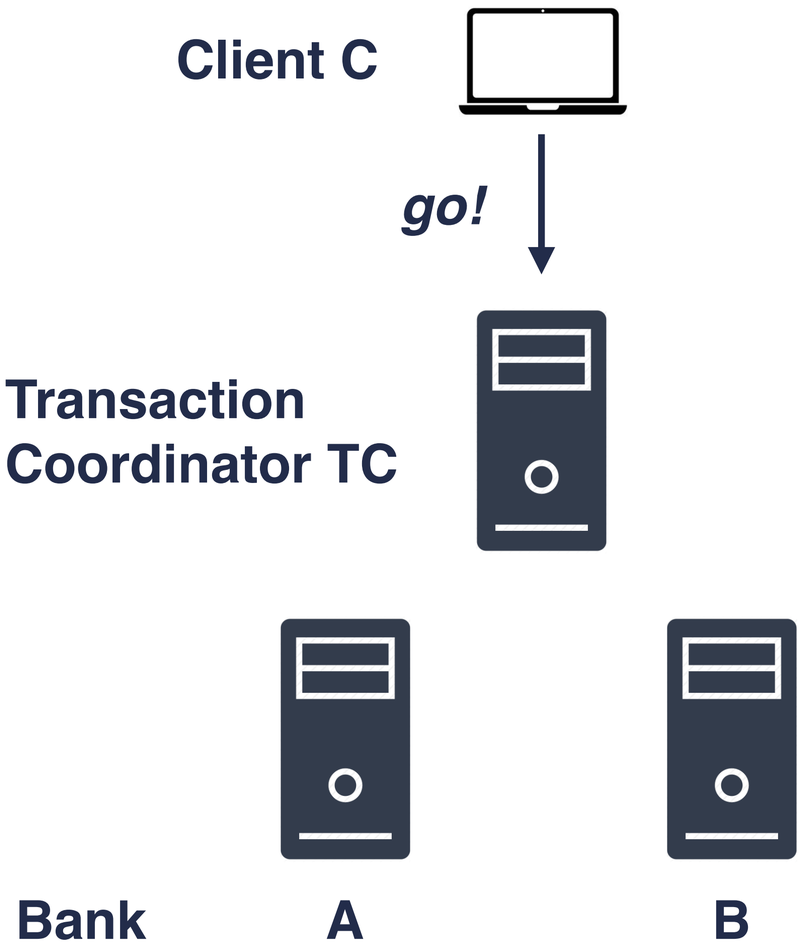
— What could possibly go wrong?

— 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

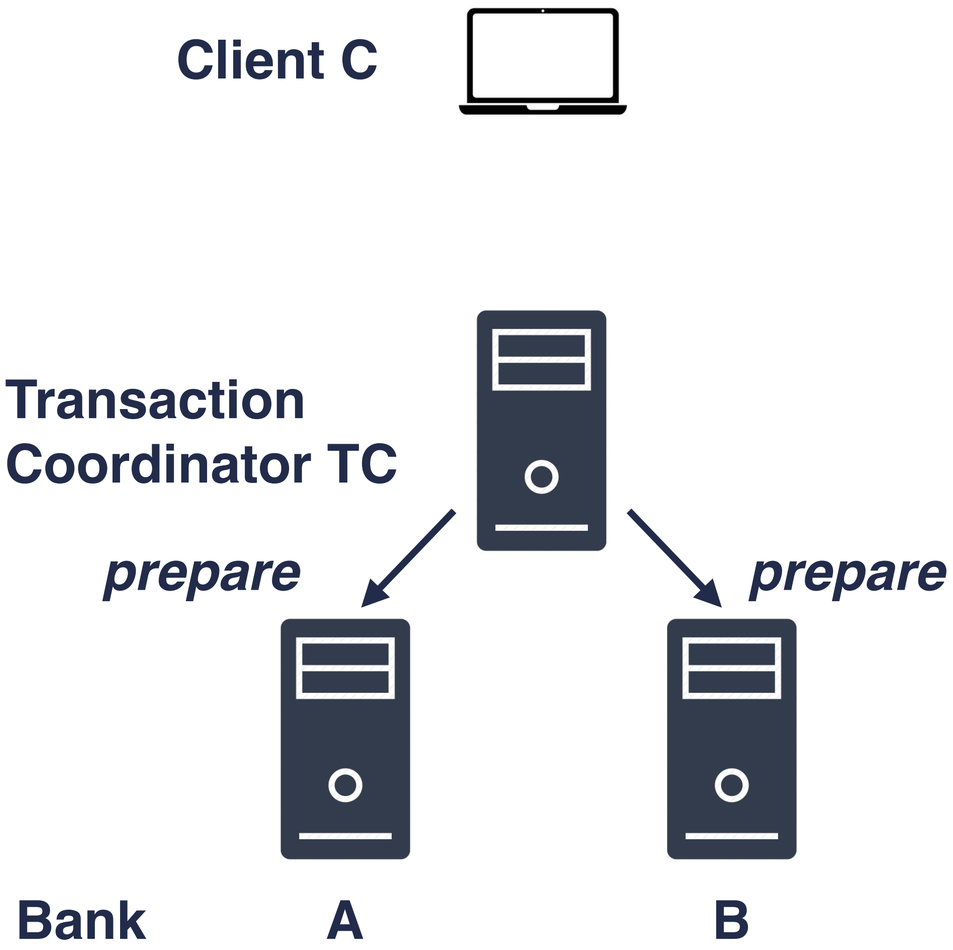
- Note that TC, A, and B each have a notion of committing
- We want two properties:
- 1. **Safety**
  - If one commits, no one aborts
  - If one aborts, no one commits
- 2. **Liveness**
  - If no failures and A and B can commit, action commits
  - If failures, reach a conclusion ASAP

A CORRECT ATOMIC COMMIT PROTOCOL



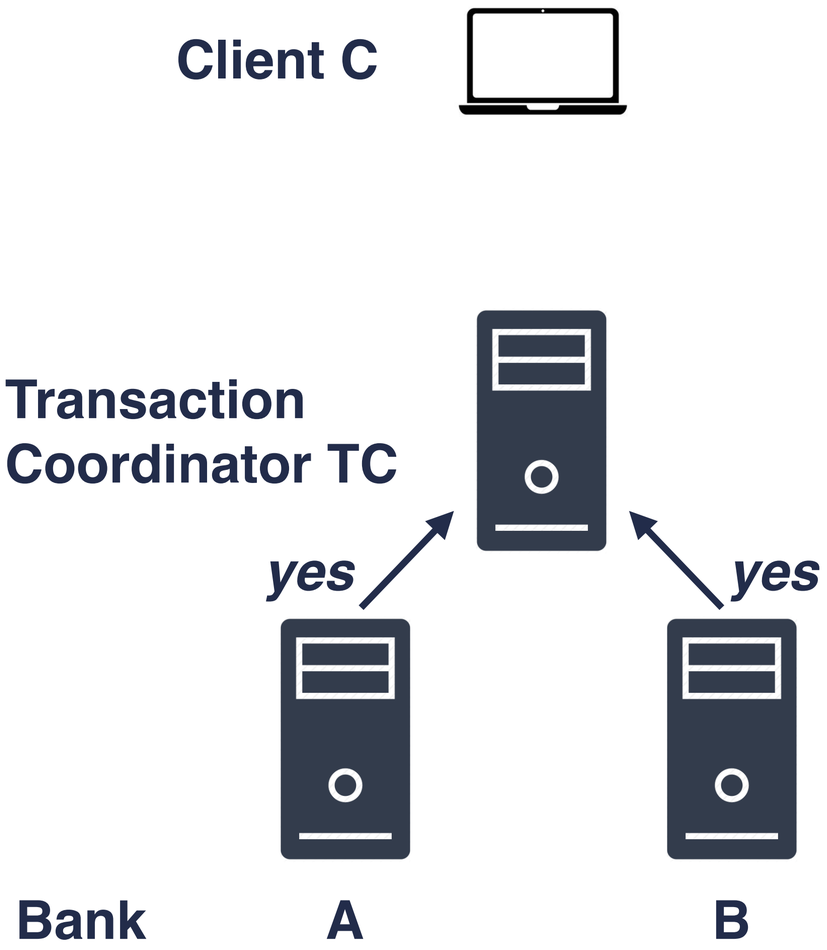
– 1. C -> TC: “go!”

A CORRECT ATOMIC COMMIT PROTOCOL



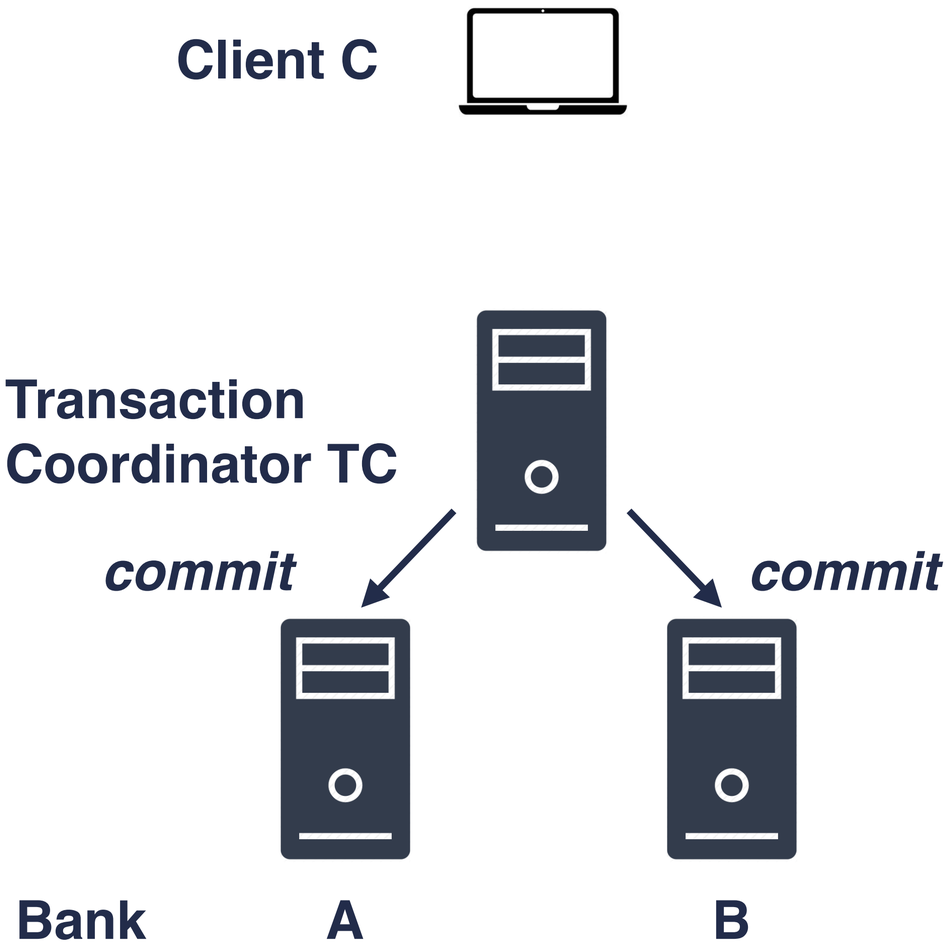
- 1. C -> TC: “go!”
- 2. TC -> A, B: “prepare!”

A CORRECT ATOMIC COMMIT PROTOCOL



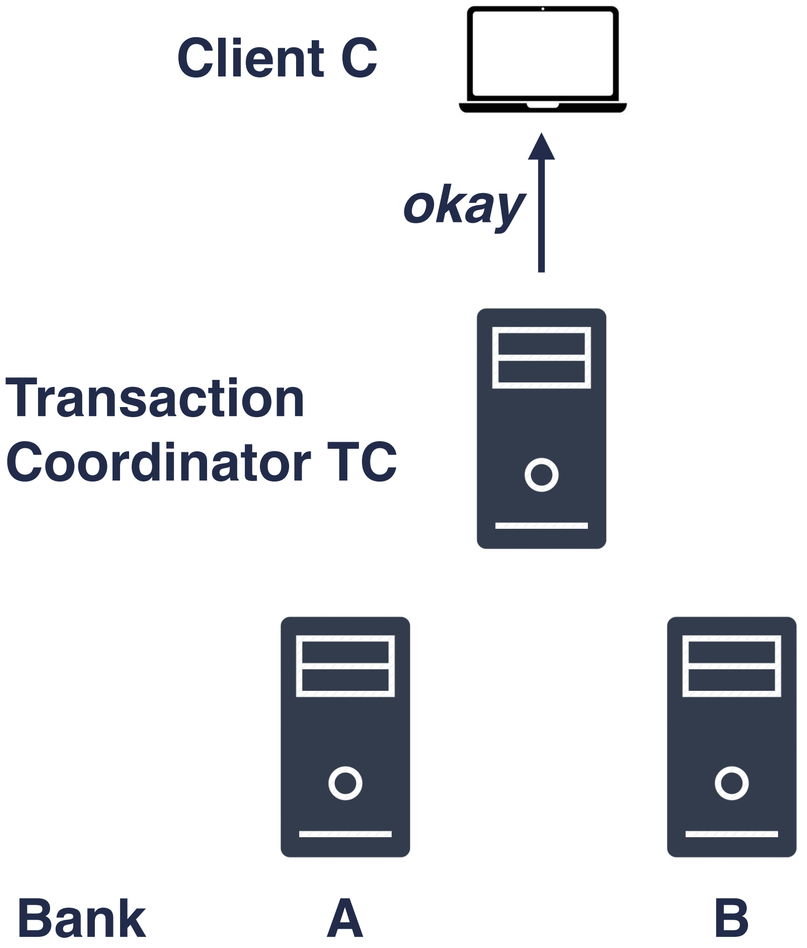
- 1. C -> TC: “go!”
- 2. TC -> A, B: “prepare!”
- 3. A, B -> P: “yes” or “no”

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
    - **Server Termination Protocol**
  - 2. Reboot: Node crashed, is rebooting, must clean up
    - **Recovery Protocol**

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):
  - Server A received commit or abort from TC:
  - Server A hasn’t voted yet or voted no:
  - Server A voted yes:

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

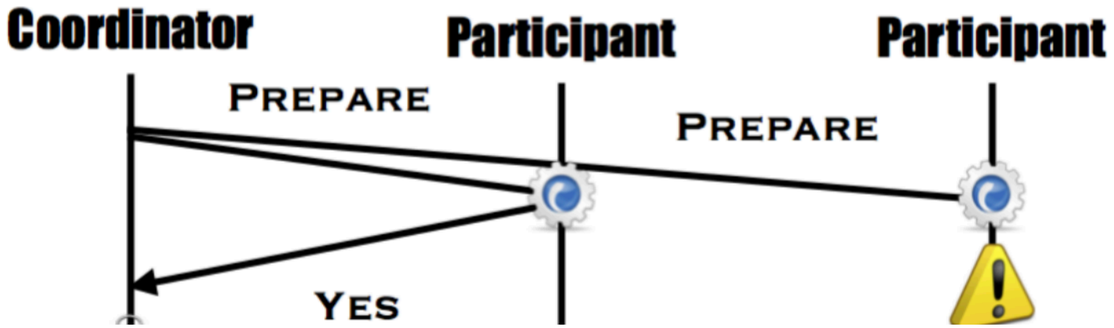
2PC Failure Scenarios

RULE

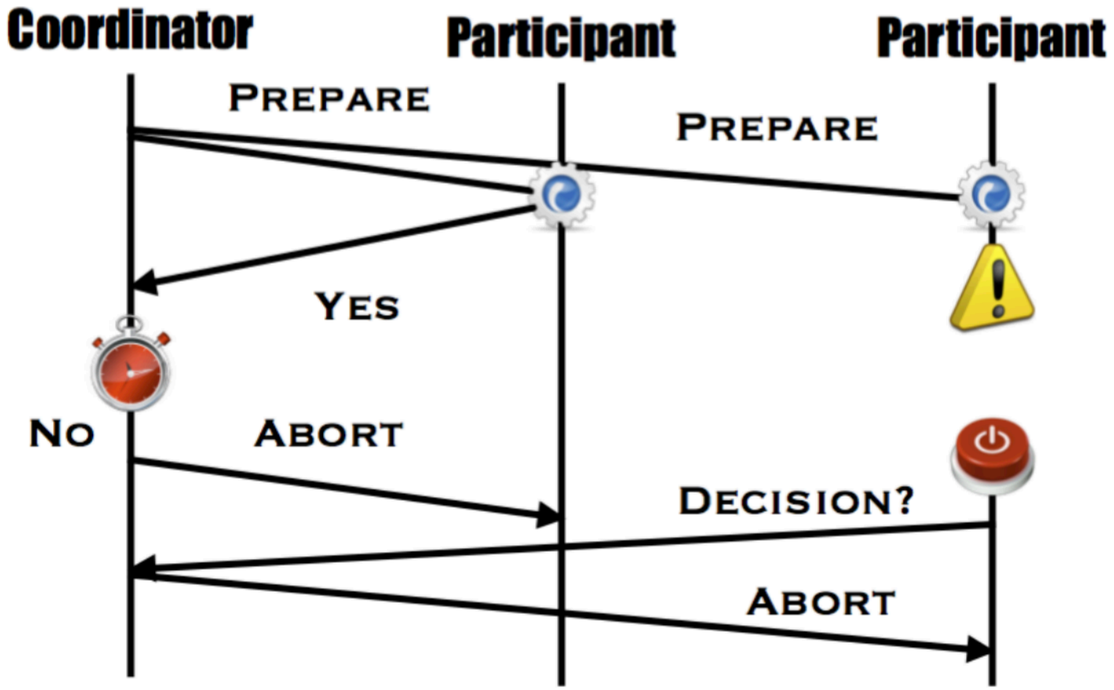
— Take a try, get one donut 

— Get it right, win an extra donut for your pal! 

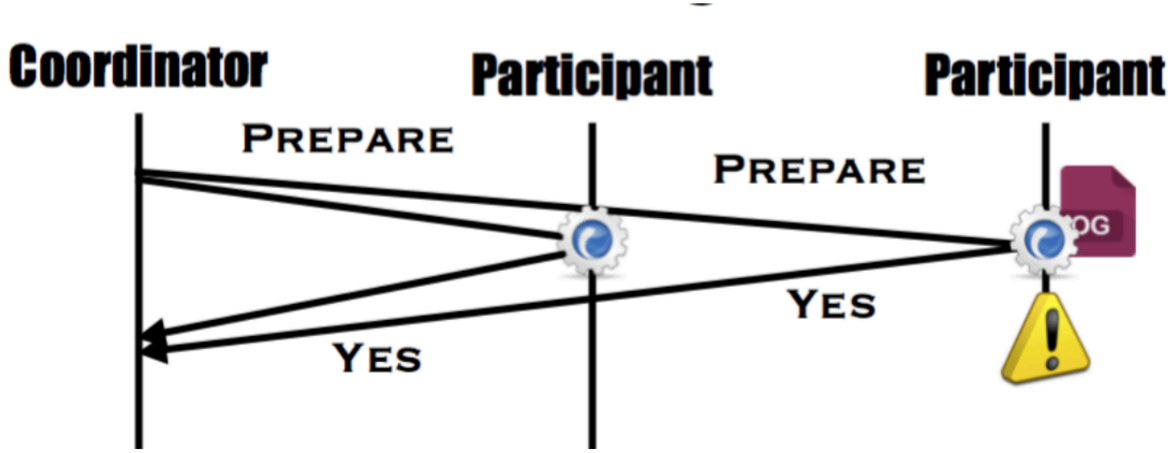
WHAT IF PARTICIPANT FAILS BEFORE SENDING RESPONSE?



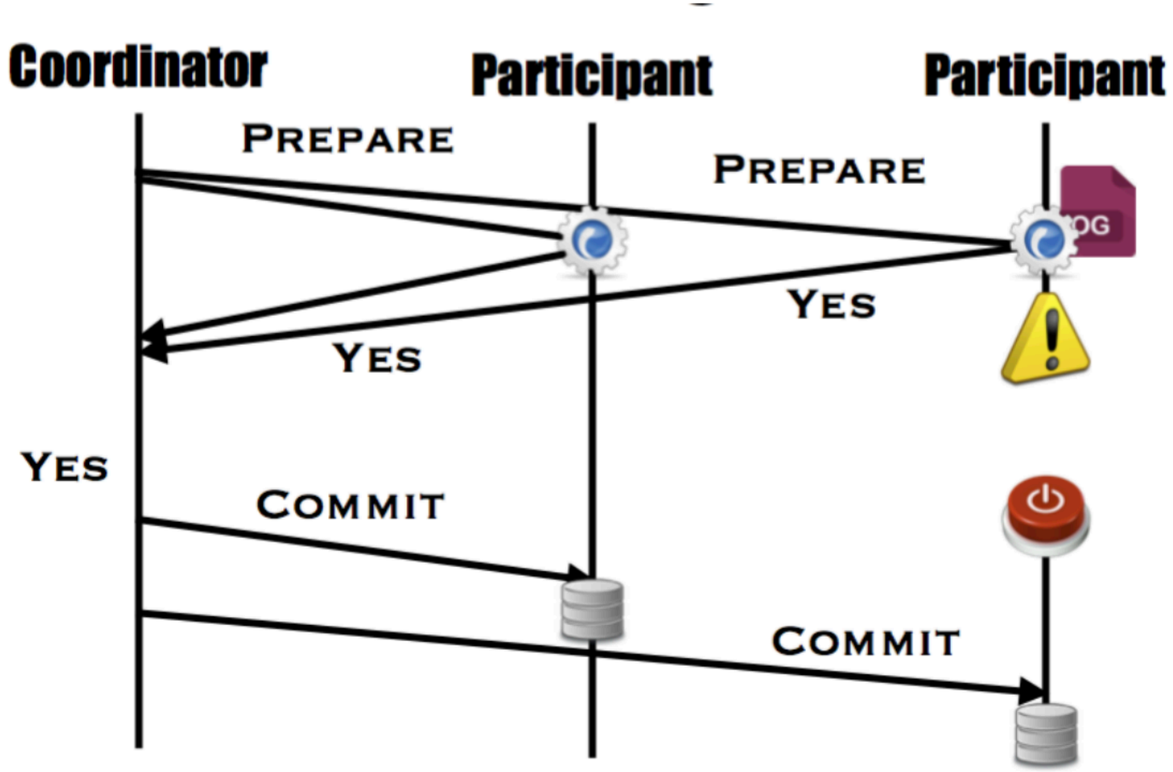
WHAT IF PARTICIPANT FAILS BEFORE SENDING RESPONSE?



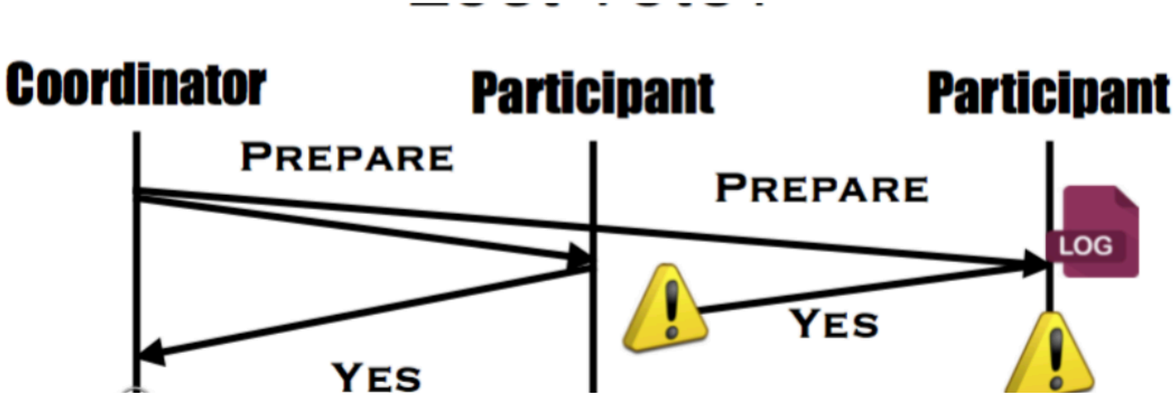
WHAT IF PARTICIPANT FAILS AFTER SENDING VOTE



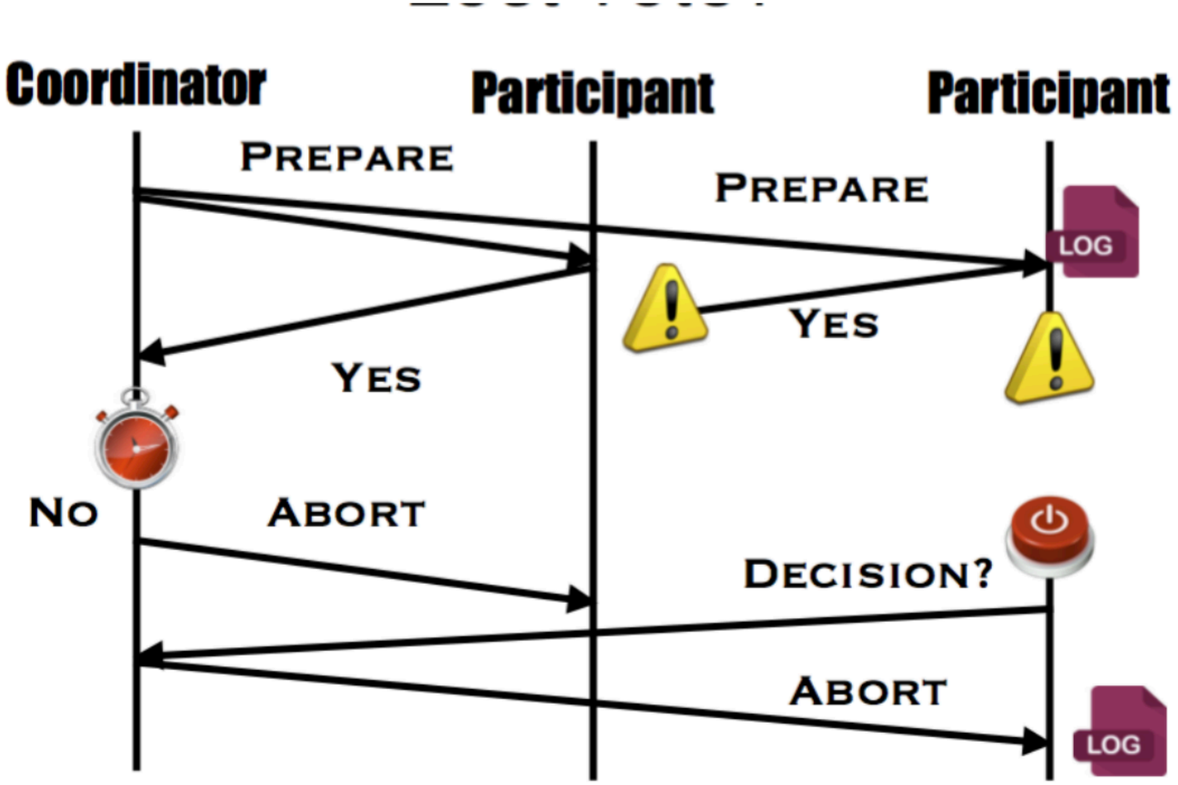
WHAT IF PARTICIPANT FAILS AFTER SENDING VOTE



WHAT IF PARTICIPANT LOST A VOTE?

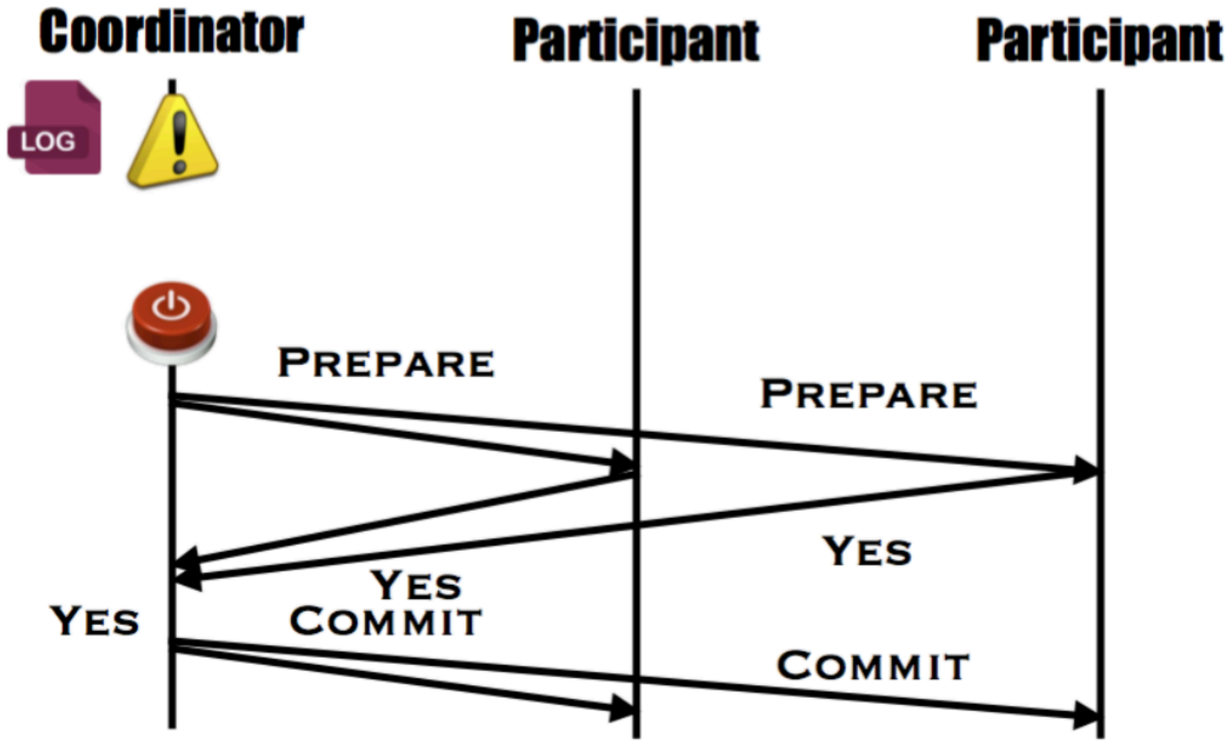


WHAT IF PARTICIPANT LOST A VOTE?

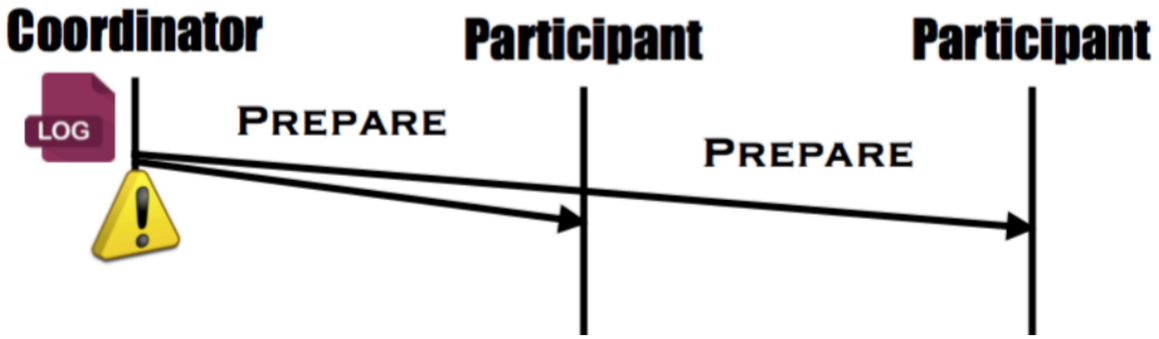


WHAT IF COORDINATOR FAILS BEFORE SENDING PREPARE?

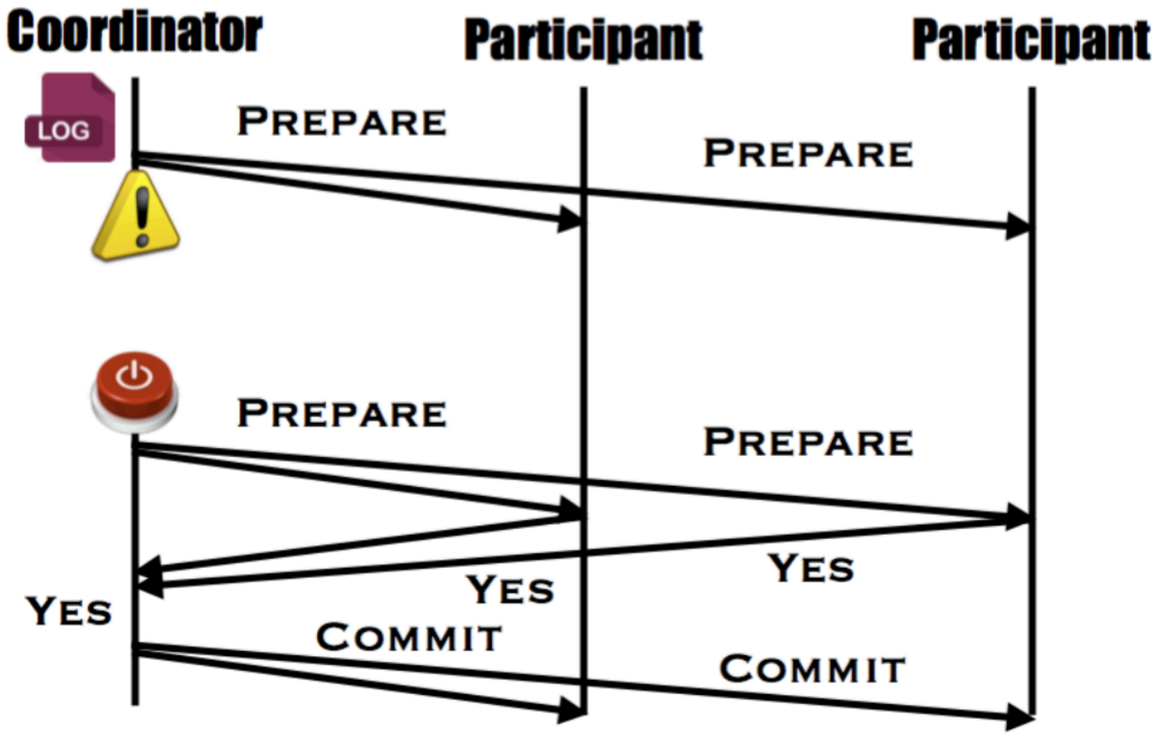
WHAT IF COORDINATOR FAILS BEFORE SENDING PREPARE?



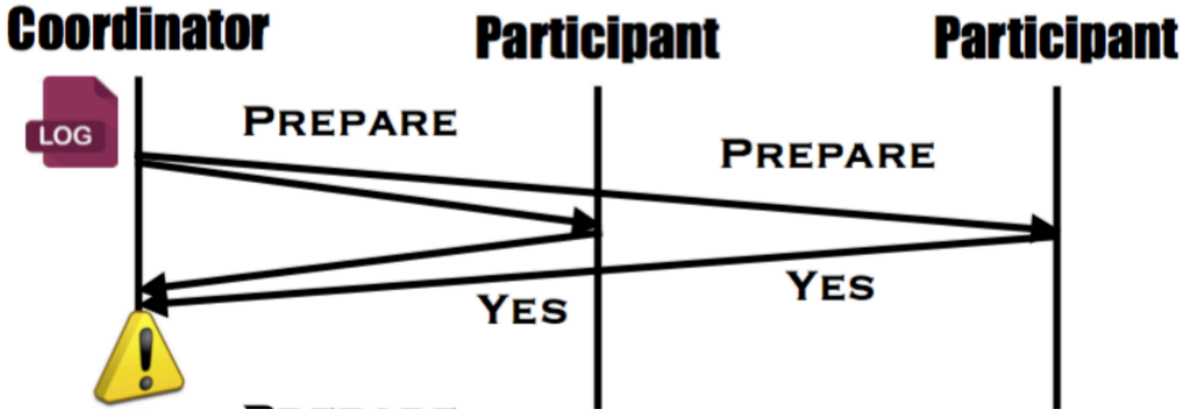
WHAT IF COORDINATOR FAILS AFTER SENDING PREPARE?



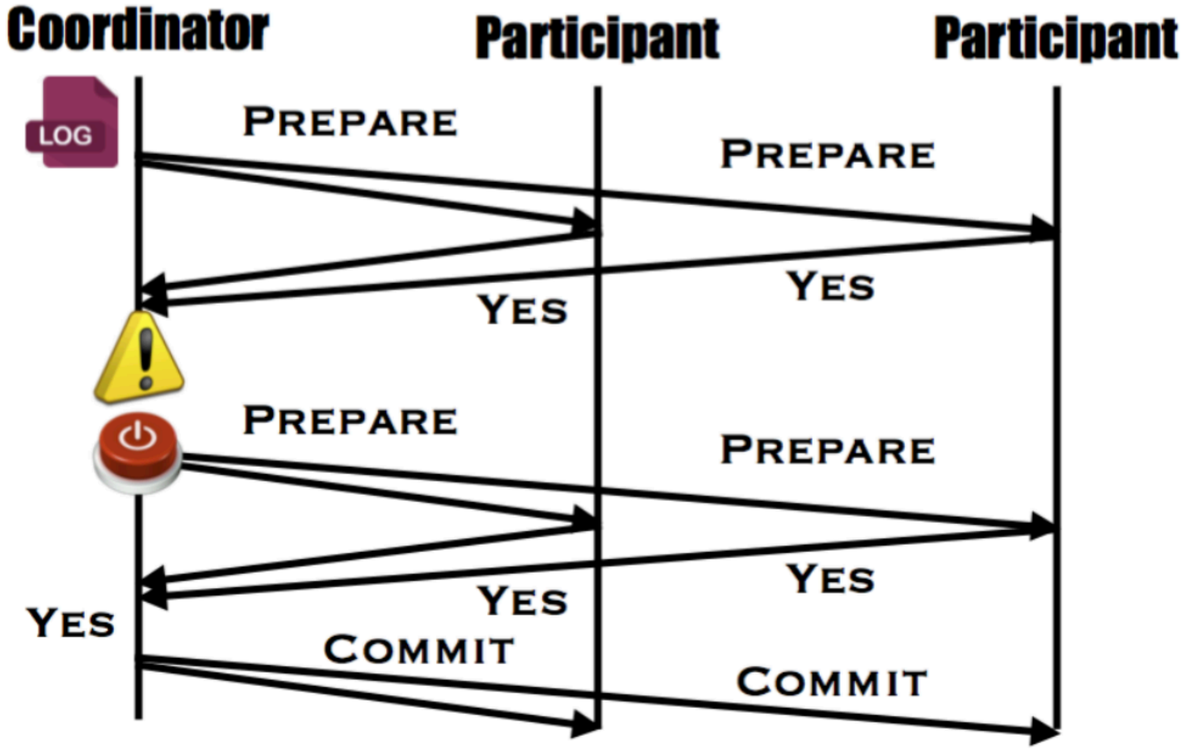
WHAT IF COORDINATOR FAILS AFTER SENDING PREPARE?



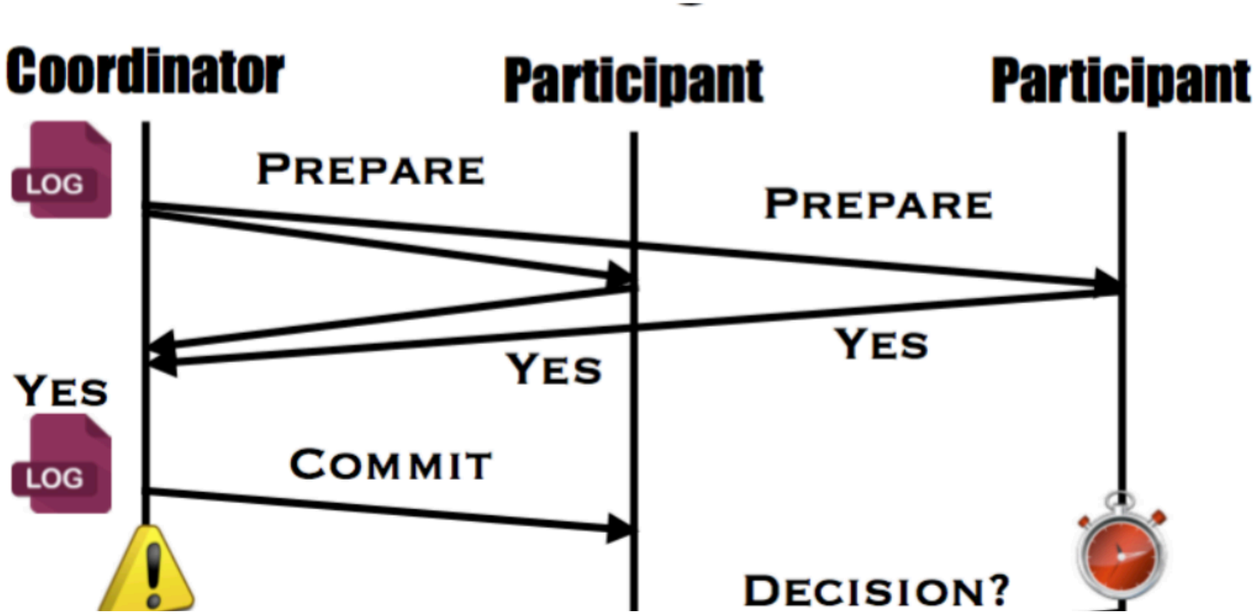
WHAT IF COORDINATOR FAILS AFTER RECEIVING VOTES



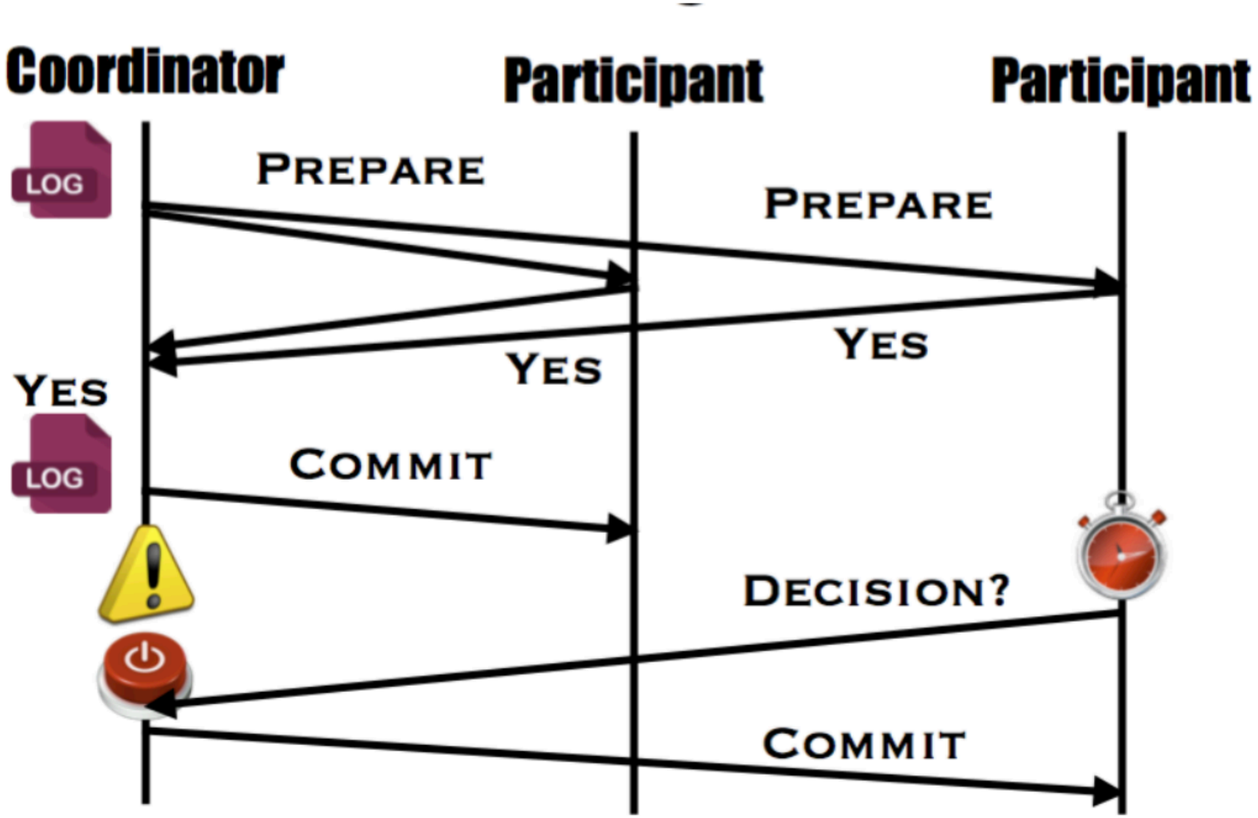
WHAT IF COORDINATOR FAILS AFTER RECEIVING VOTES



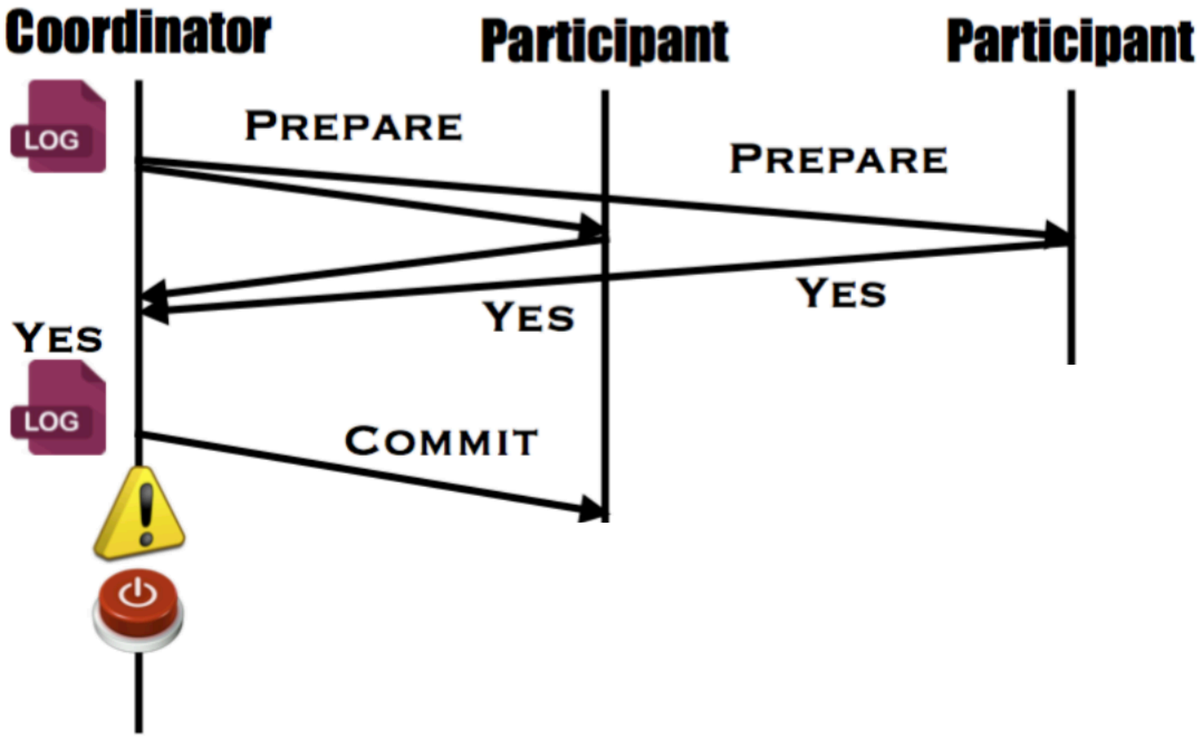
WHAT IF COORDINATOR FAILS AFTER SENDING DECISION?



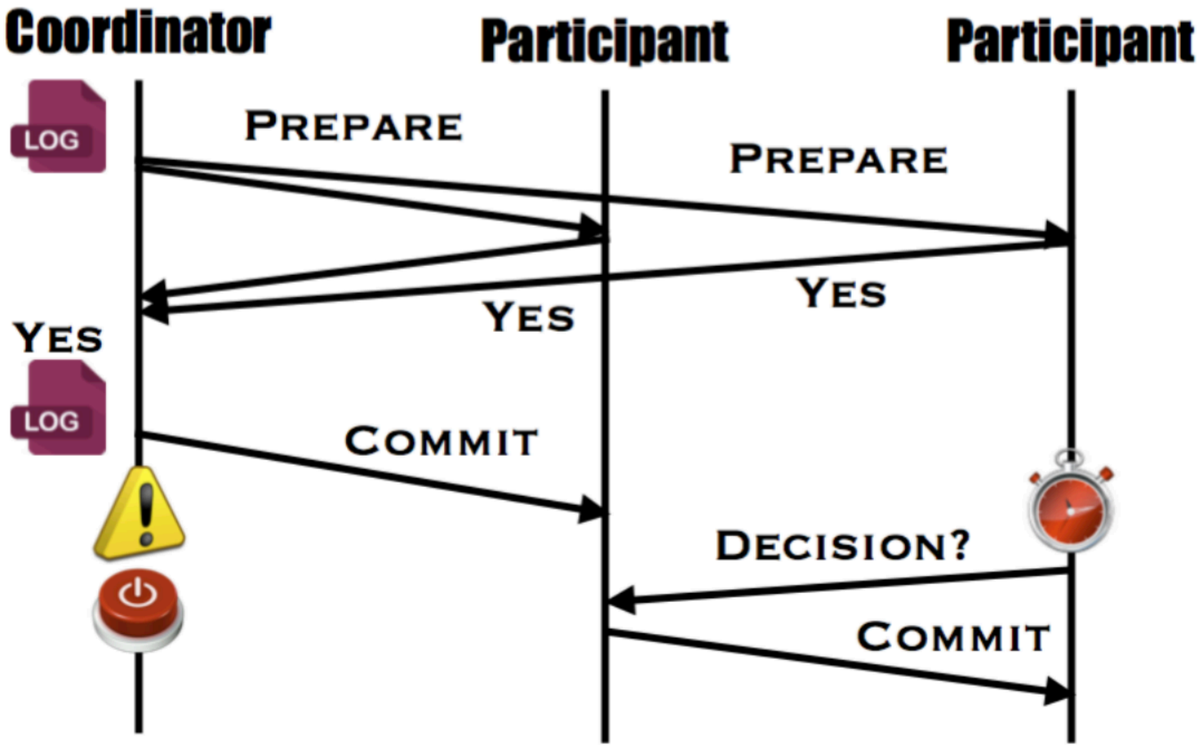
WHAT IF COORDINATOR FAILS AFTER SENDING DECISION?



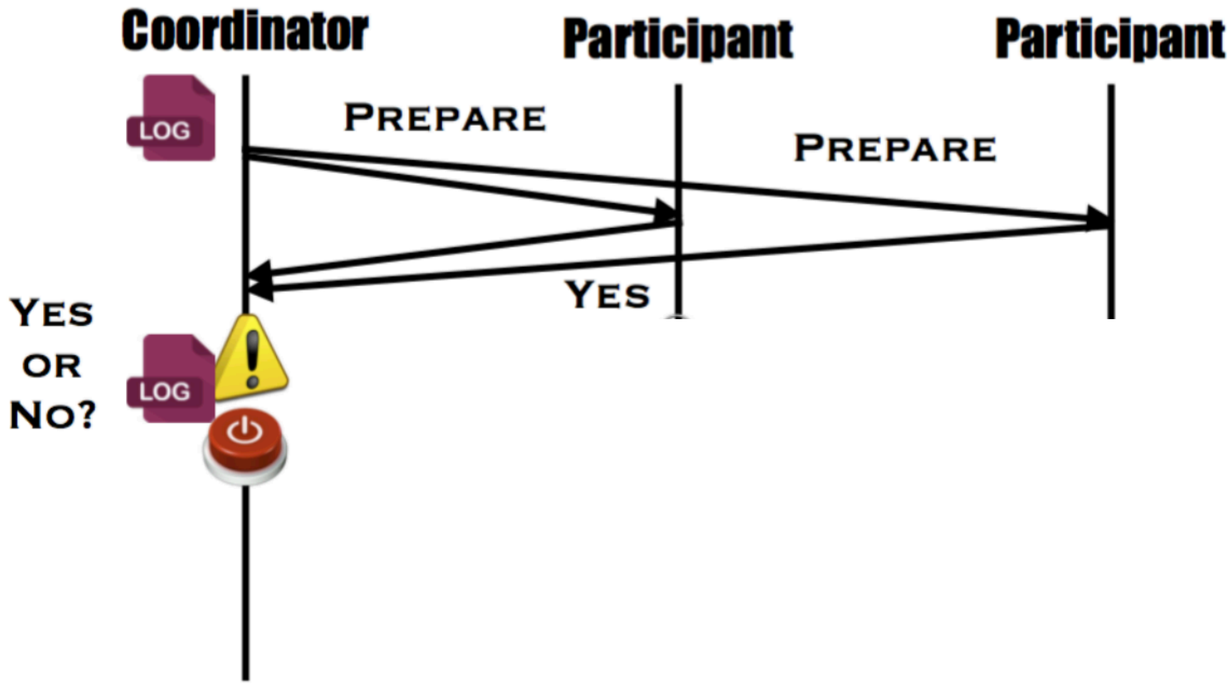
WHAT IF COORDINATOR FAILS AFTER SENDING ONLY ONE DECISION?



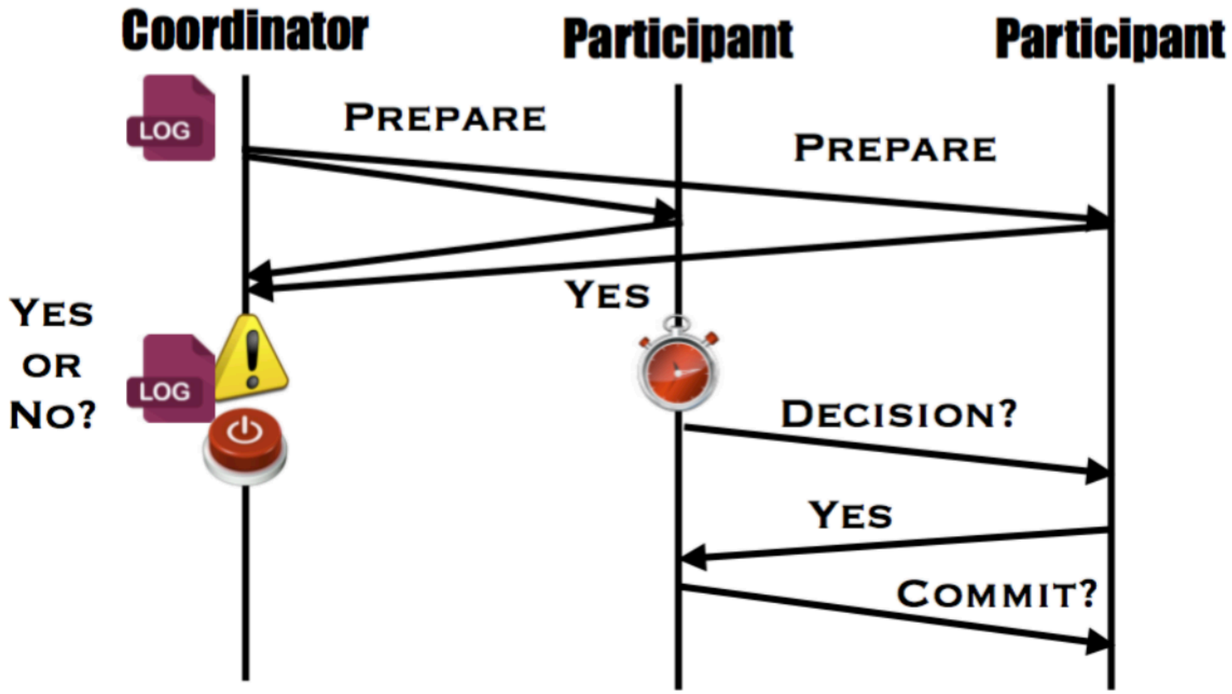
WHAT IF COORDINATOR FAILS AFTER SENDING ONLY ONE DECISION?



WHAT IF COORDINATOR FAILS BEFORE SENDING ANY DECISION?



WHAT IF COORDINATOR FAILS BEFORE SENDING ANY DECISION?



2PC Limitations

2PC IS BLOCKING

- A process can block indefinitely in its uncertainty period until a TC or network failure is resolved.
- If TC is also a participant, then a single-site failure can cause 2PC to block indefinitely!
- And it blocks while each shard server is holding locks, preventing other transactions that don’t even interact with the failed shard server from making progress!
- This is why 2PC is called a blocking protocol and cannot be used as a basis for fault tolerance.

2PC IS EXPENSIVE

- Time complexity: 3 message latencies on the critical path: PREPARE → PREPARE-OK/FAIL → ABORT/COMMIT.
- Message complexity: common case for n participants + 1 TC: 3n messages.
- That's expensive, esp. if shards are geo distributed.
- Optimizations, or adding an extra phase (3PC), cannot address the blocking/performance problems of 2PC while maintaining its semantic.

UNIVERSITY of VIRGINIA

TAKEAWAYS

- How to construct our fuller Web service architecture from Lecture 1 (took us long, ha)?
  - today: how to achieve scalability through sharding, with 2PC.
  - future: how to achieve fault tolerance through replication.
- Next class: Guest Talk

ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.