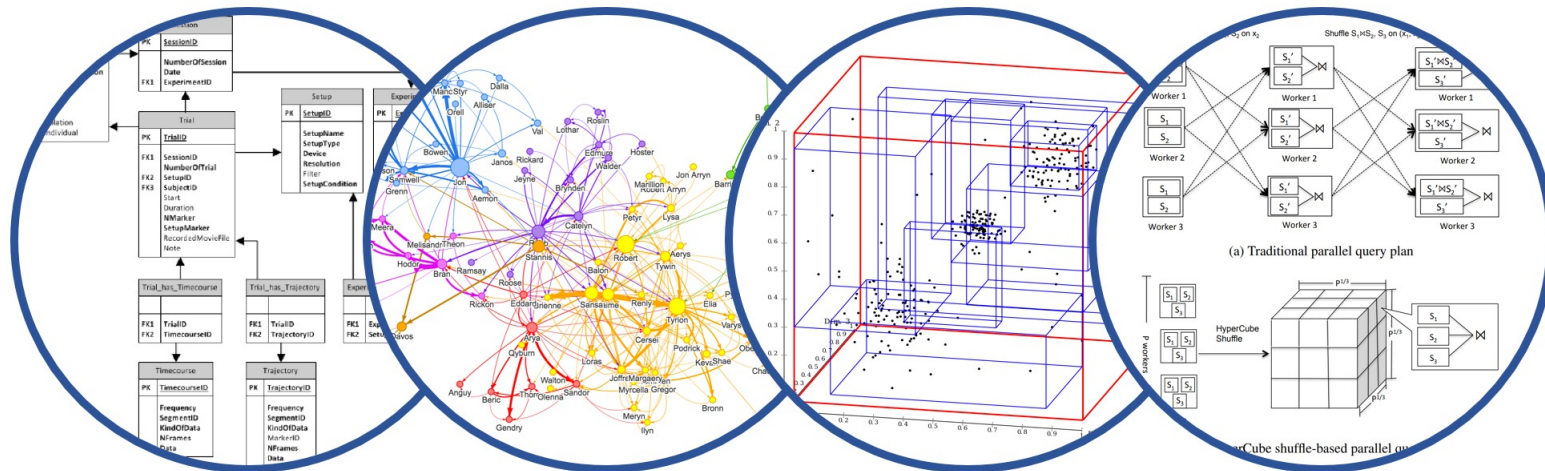


# Course evals (first 5 minutes)

Please take a few minutes to fill out the course evaluations:

<https://uw.iasystem.org/survey/324590>

And thank you all for your hard work this quarter!



# Database System Internals

## Two-Phase Commit (2PC)

Paul G. Allen School of Computer Science and Engineering  
University of Washington, Seattle

# References

- Ullman book: Section 20.5
- Ramakrishnan book: Chapter 22

# We are Learning about Scaling DBMSs

- **Scaling the execution of a query**

- ↳ Parallel DBMS
- ↳ MapReduce
- ↳ Spark

- **Scaling transactions**



- Distributed transactions
- Replication
- Scaling with NoSQL and NewSQL

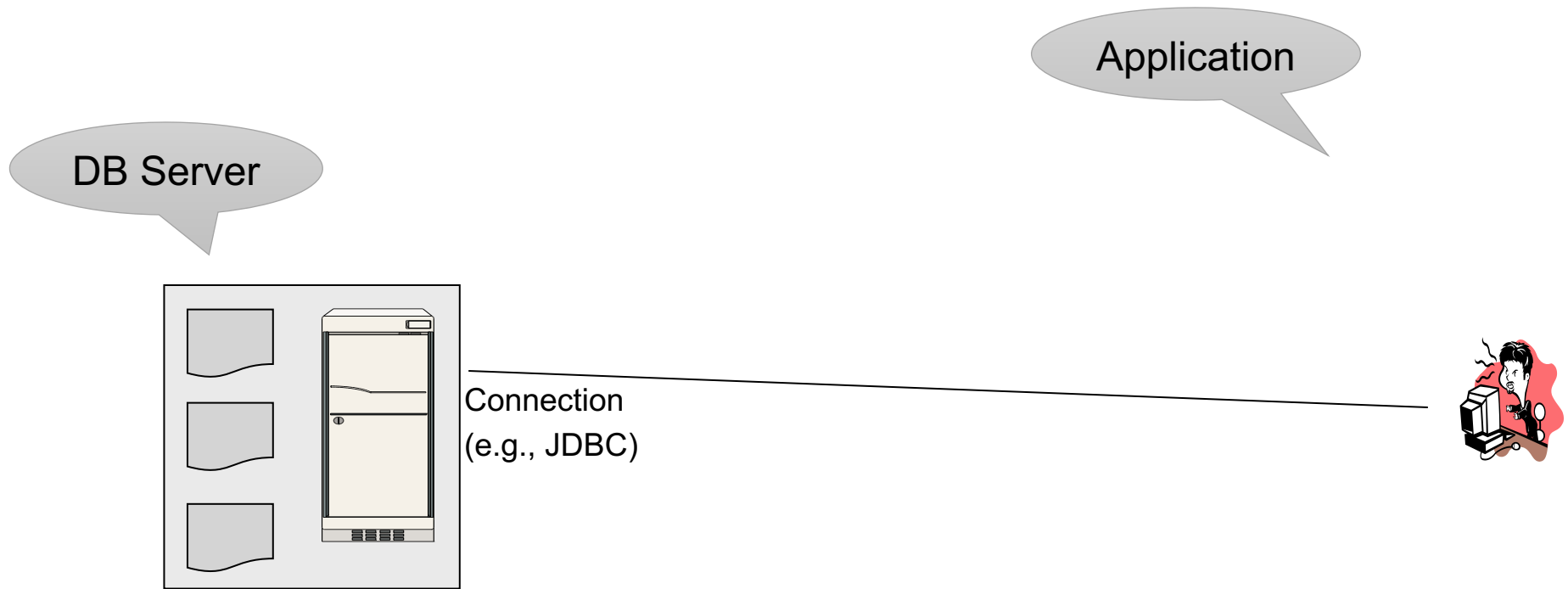
# Scaling Transactions Per Second

- OLTP: Transactions per second  
“Online Transaction Processing”
- Amazon
- Facebook
- Twitter
- ... your favorite Internet application...
- Goal is to increase transaction throughput

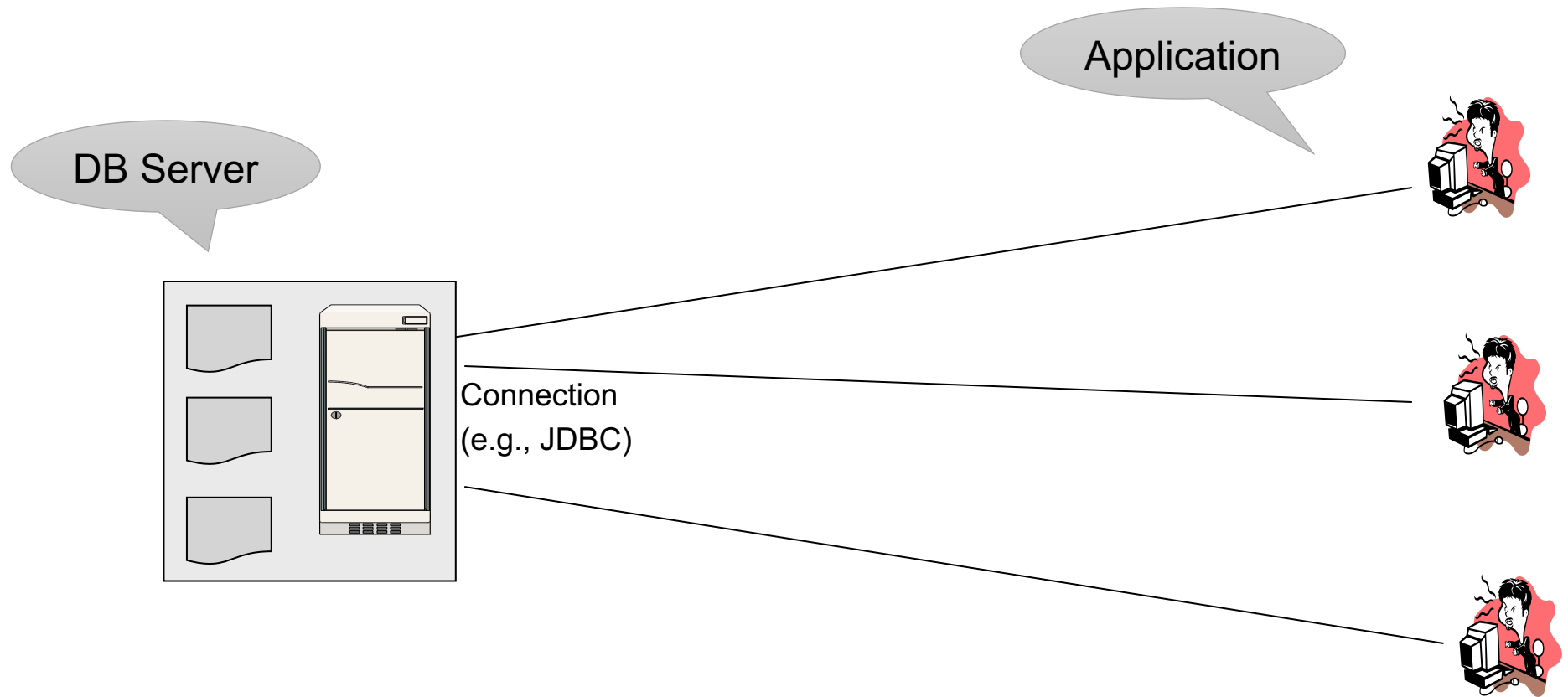
# How to Scale the DBMS?

- Can easily replicate the web servers and the application servers
- We cannot so easily replicate the database servers, because the database is unique
- We need to design ways to **scale up the DBMS**

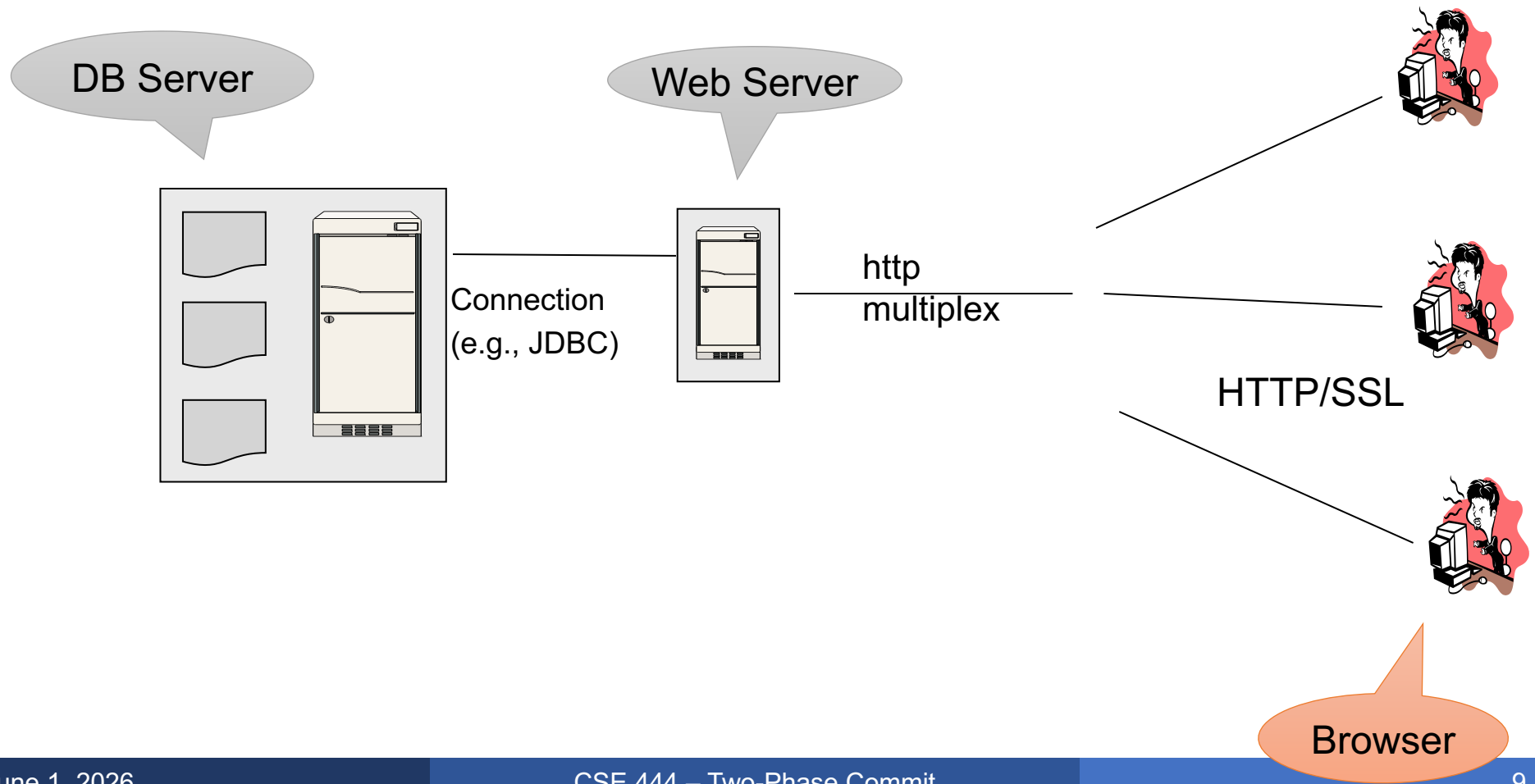
# How to Scale?



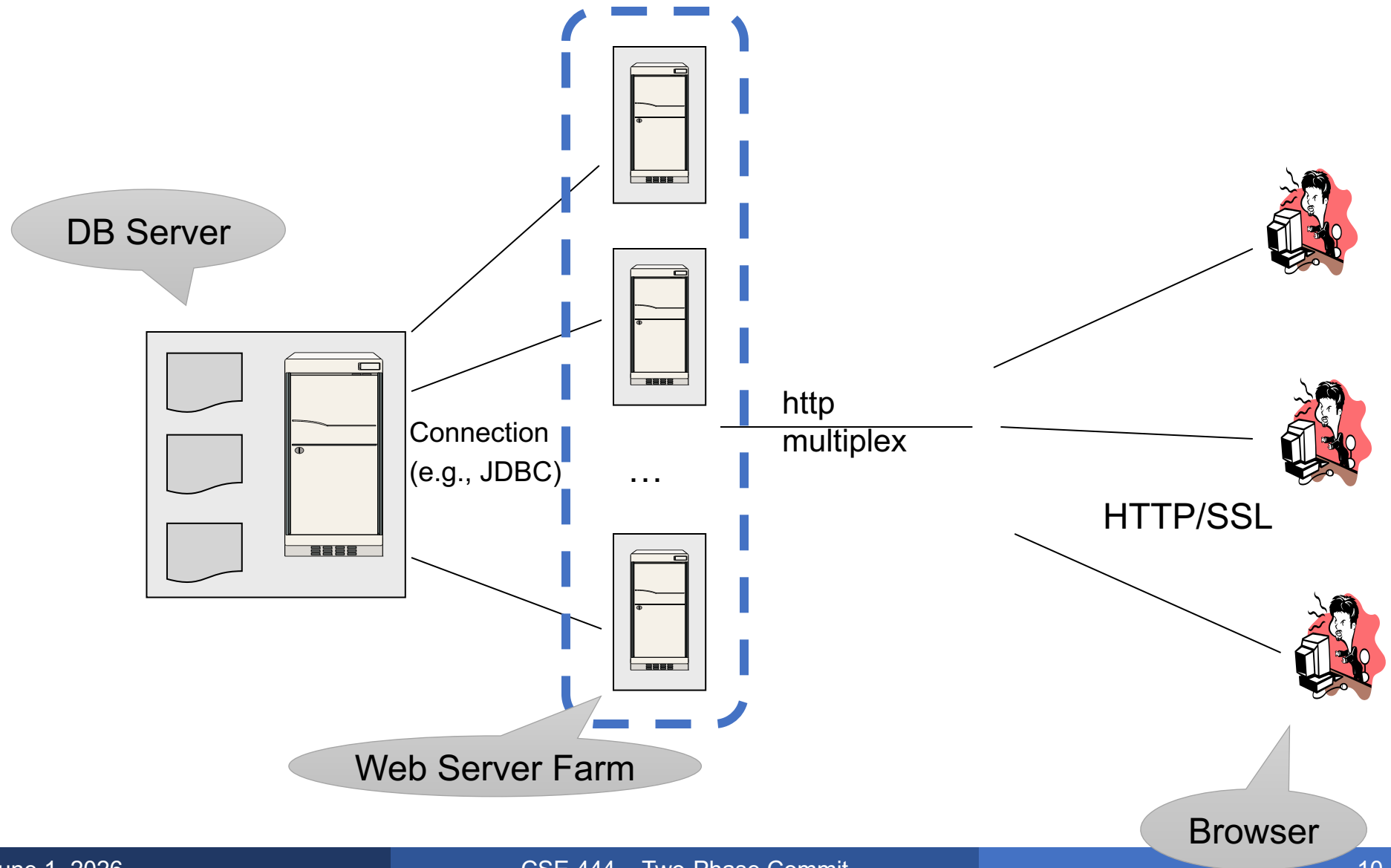
# How to Scale?



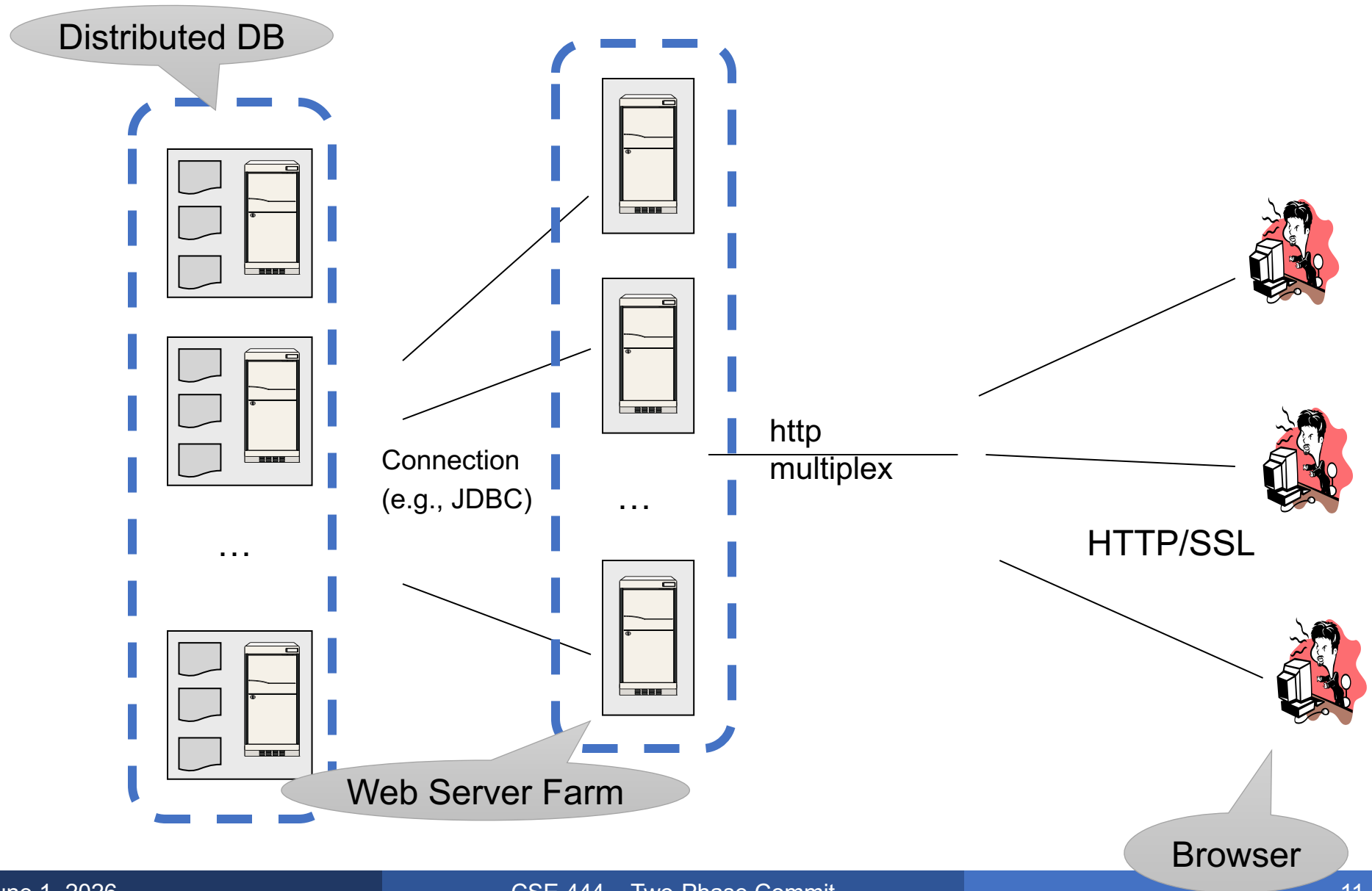
# How to Scale?



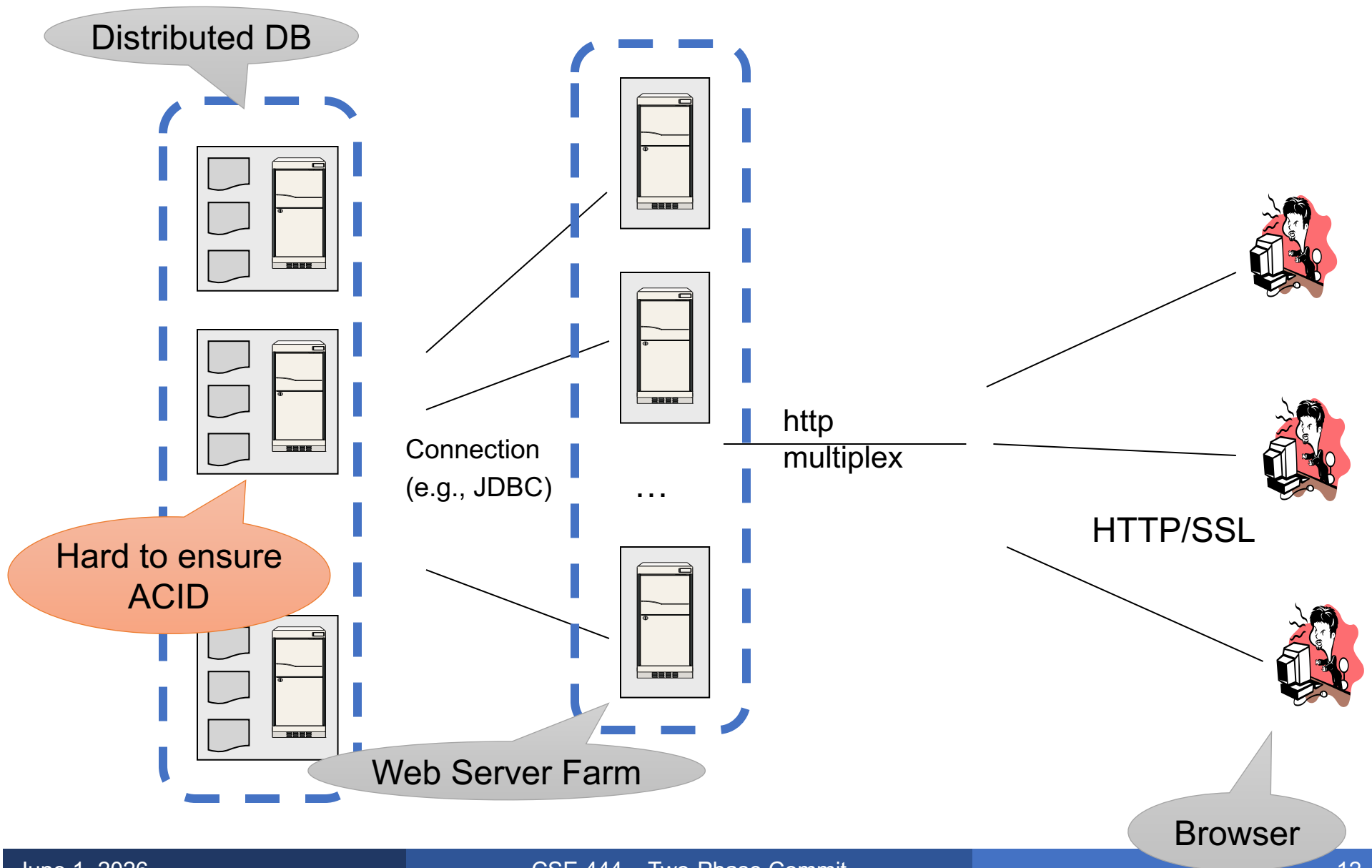
# How to Scale?



# How to Scale?



# How to Scale?



# Transaction Scaling Challenges

## ■ Distribution

- There is a limit on transactions/sec on one server
- Need to partition the database across multiple servers
- If a transaction touches one machine, life is good!
- If a transaction touches multiple machines, ACID becomes extremely expensive! Need two-phase commit

## ■ Replication

- Replication can help to increase throughput and lower latency
- Create multiple copies of each database partition
- Spread queries across these replicas
- Easy for reads but writes, once again, become expensive!

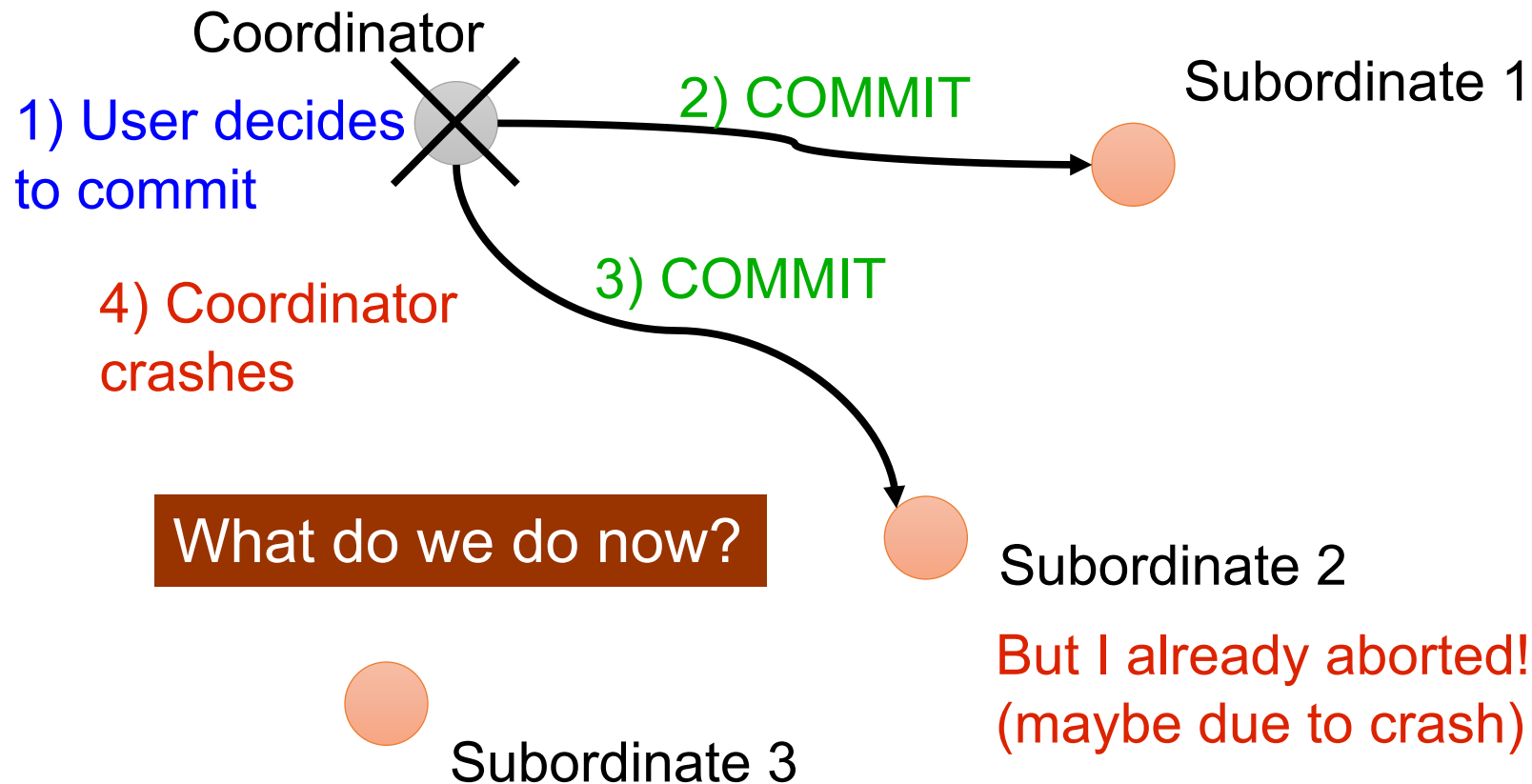
# Distributed Transactions

- Concurrency control
- Failure recovery
  - Transaction must be committed at all sites or at none of the sites!
  - No matter what failures occur and when they occur
  - Two-phase commit protocol (2PC)

# Distributed Concurrency Control

- In theory, different techniques are possible
  - Pessimistic, optimistic, locking, timestamps
- In practice, distributed two-phase locking
  - Simultaneously hold locks at all sites involved
- Deadlock detection techniques
  - Global wait-for graph (not very practical)
  - Timeouts
- If deadlock: abort least costly local transaction

# Two-Phase Commit: Motivation



# 2PC Outline

- Phase 1: coordinator polls the subordinators whether they want to commit or abort
- Phase 2: coordinator notifies all subordinators of the decision commit or abort

# 2PC: Phase 1, Prepare

Coordinator



Subordinate 1



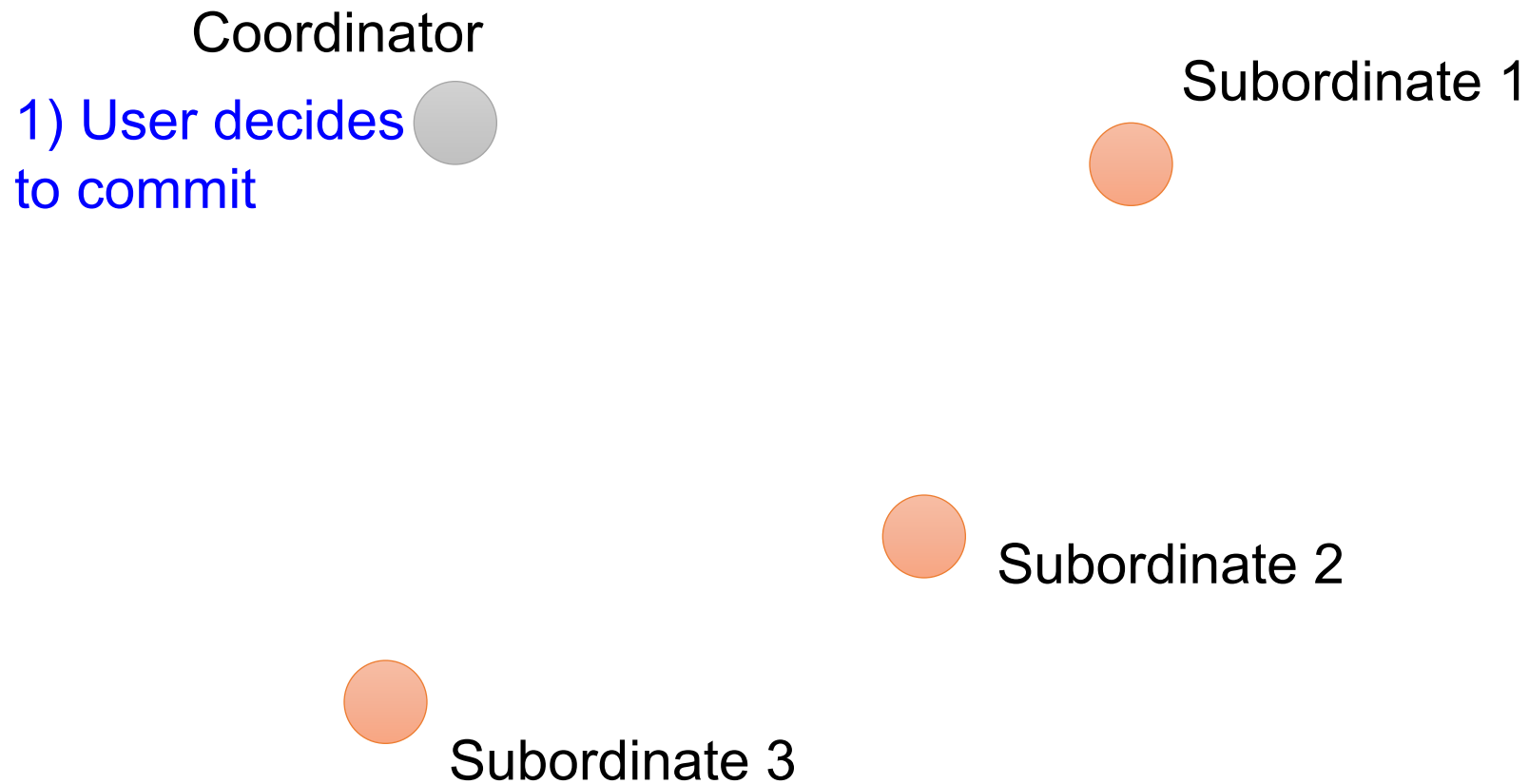
Subordinate 2



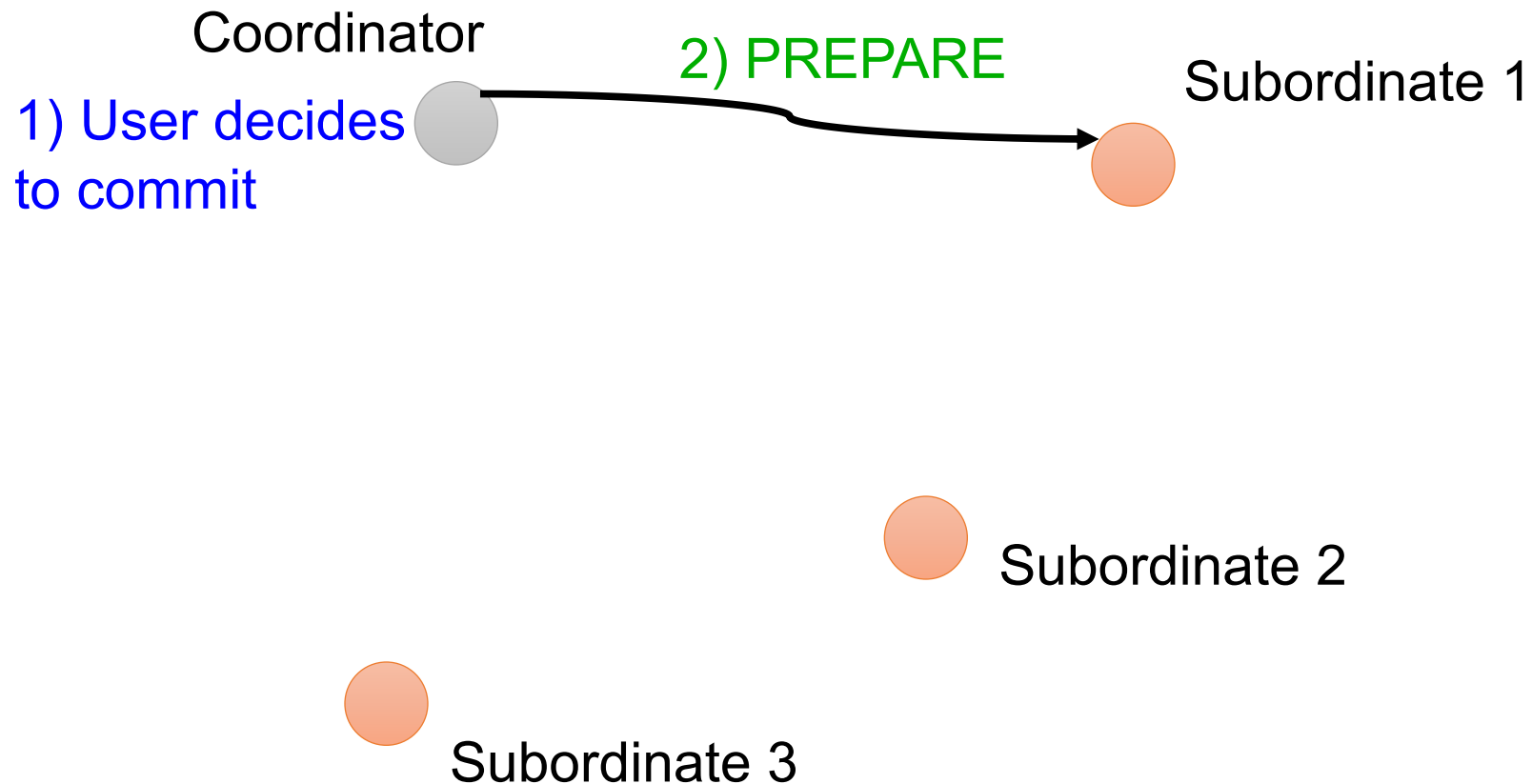
Subordinate 3



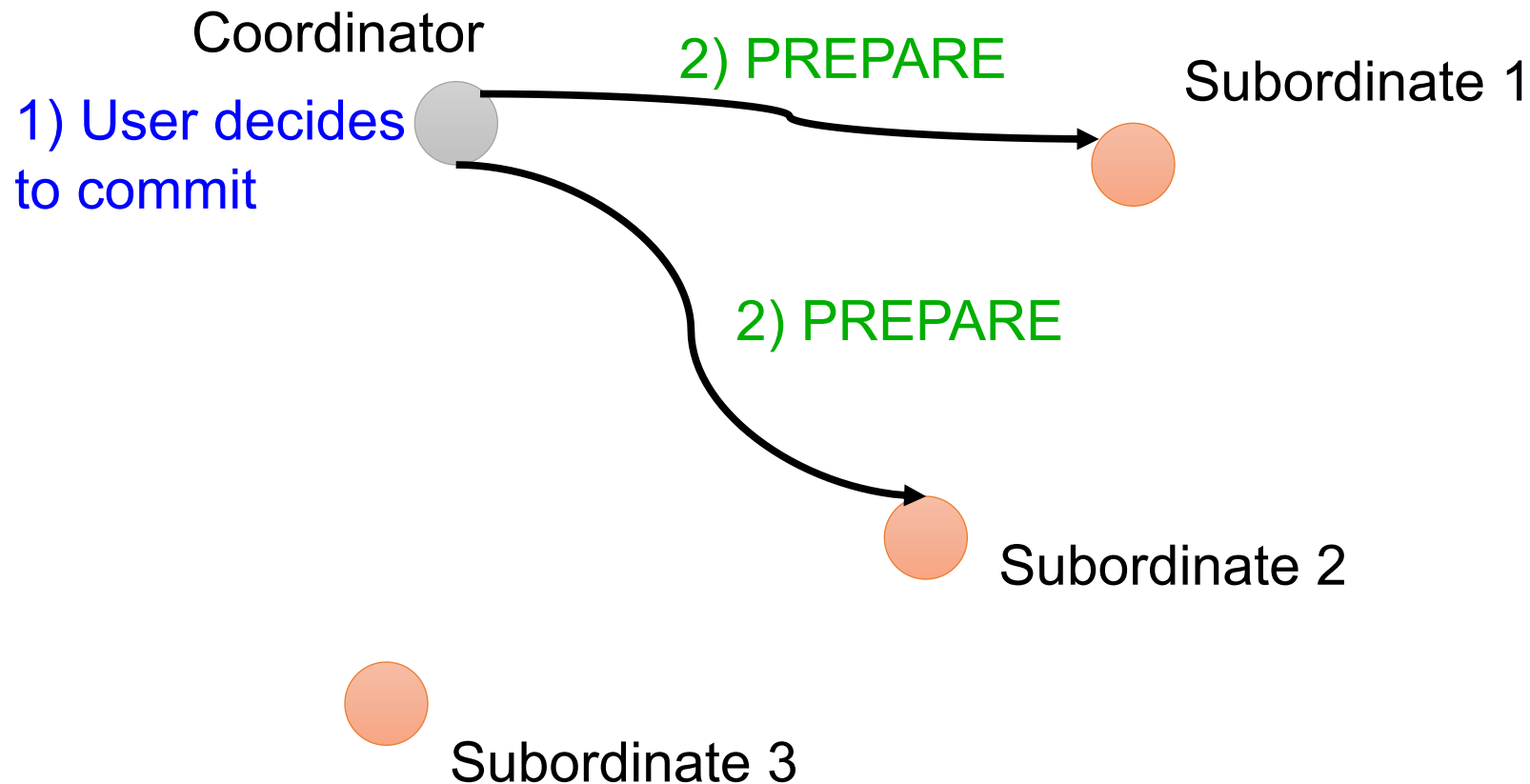
# 2PC: Phase 1, Prepare



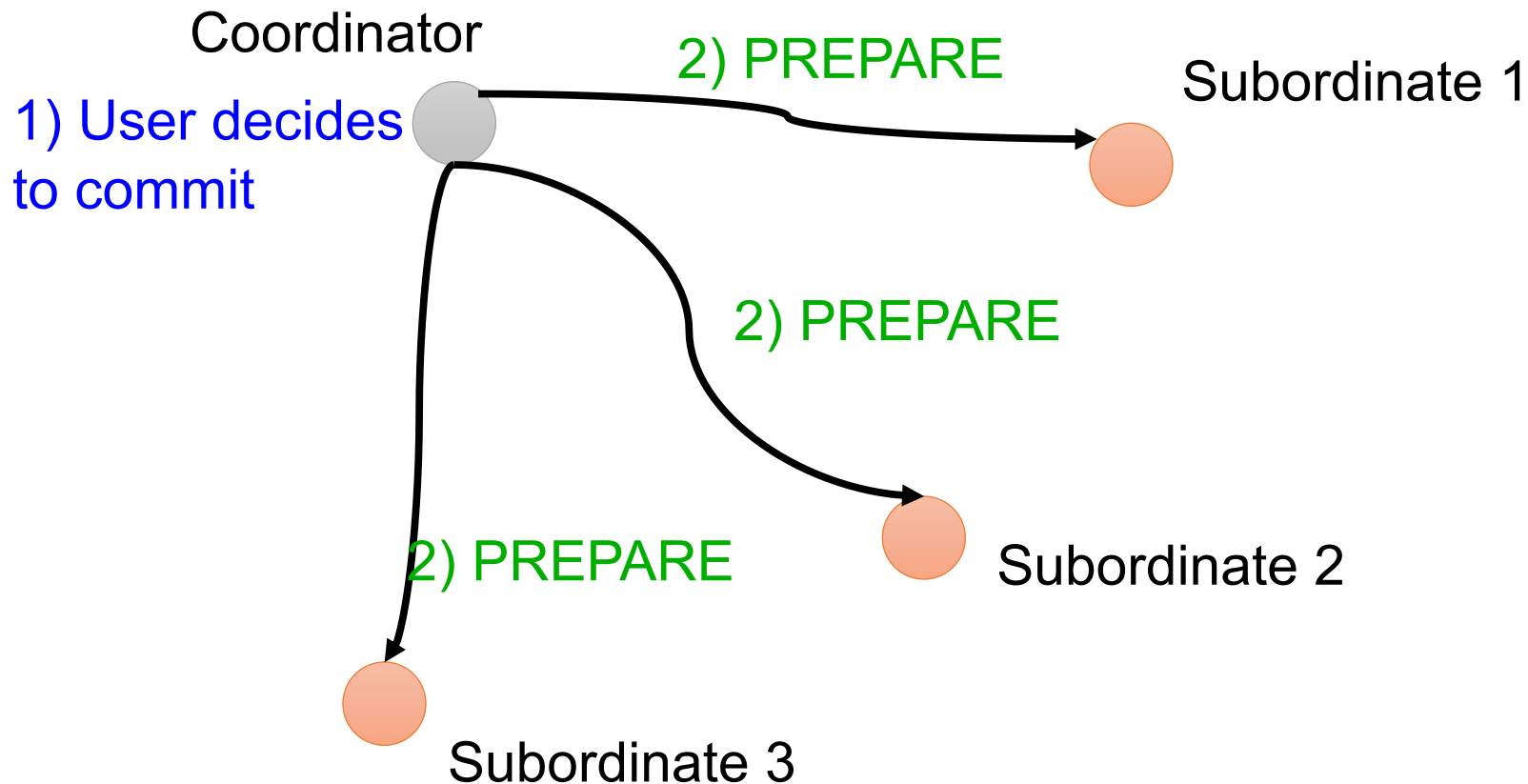
# 2PC: Phase 1, Prepare



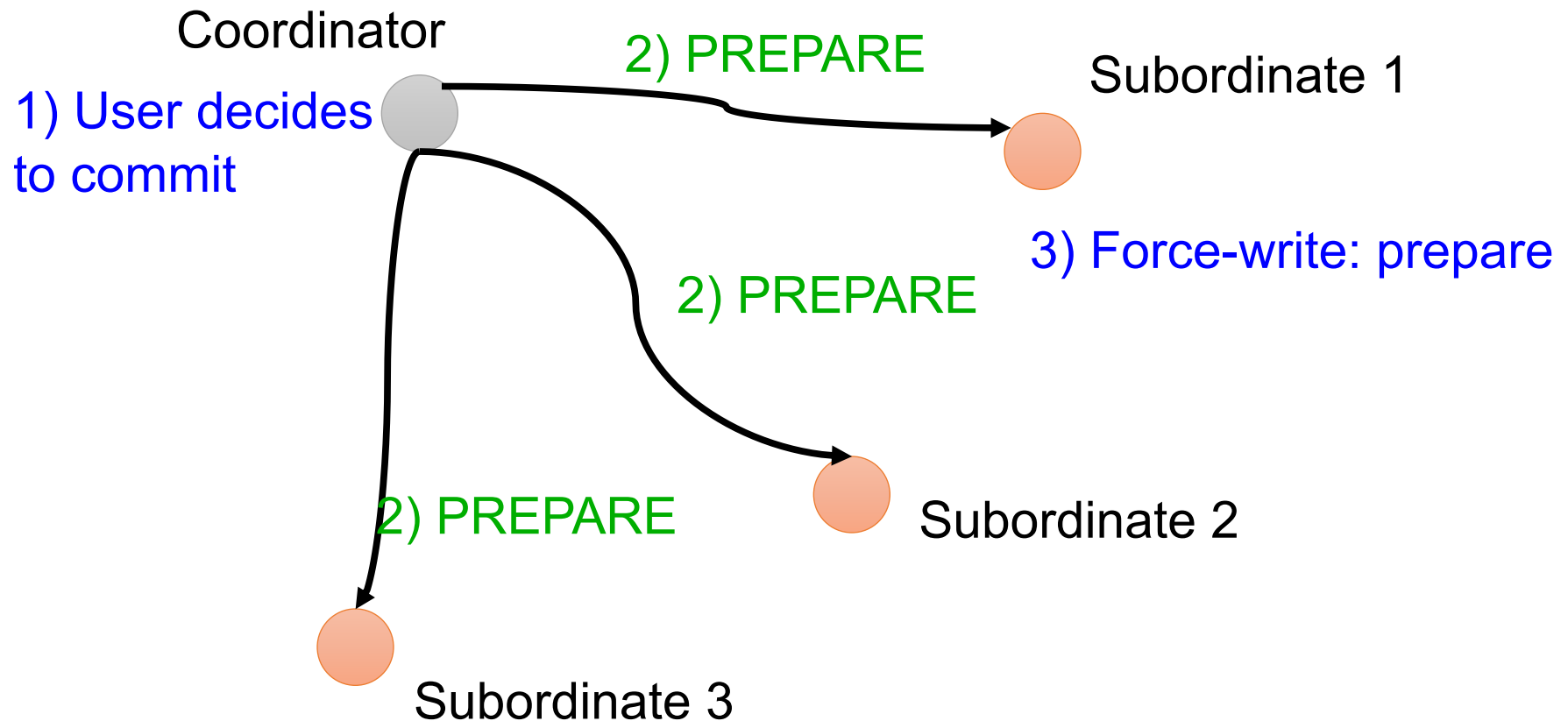
# 2PC: Phase 1, Prepare



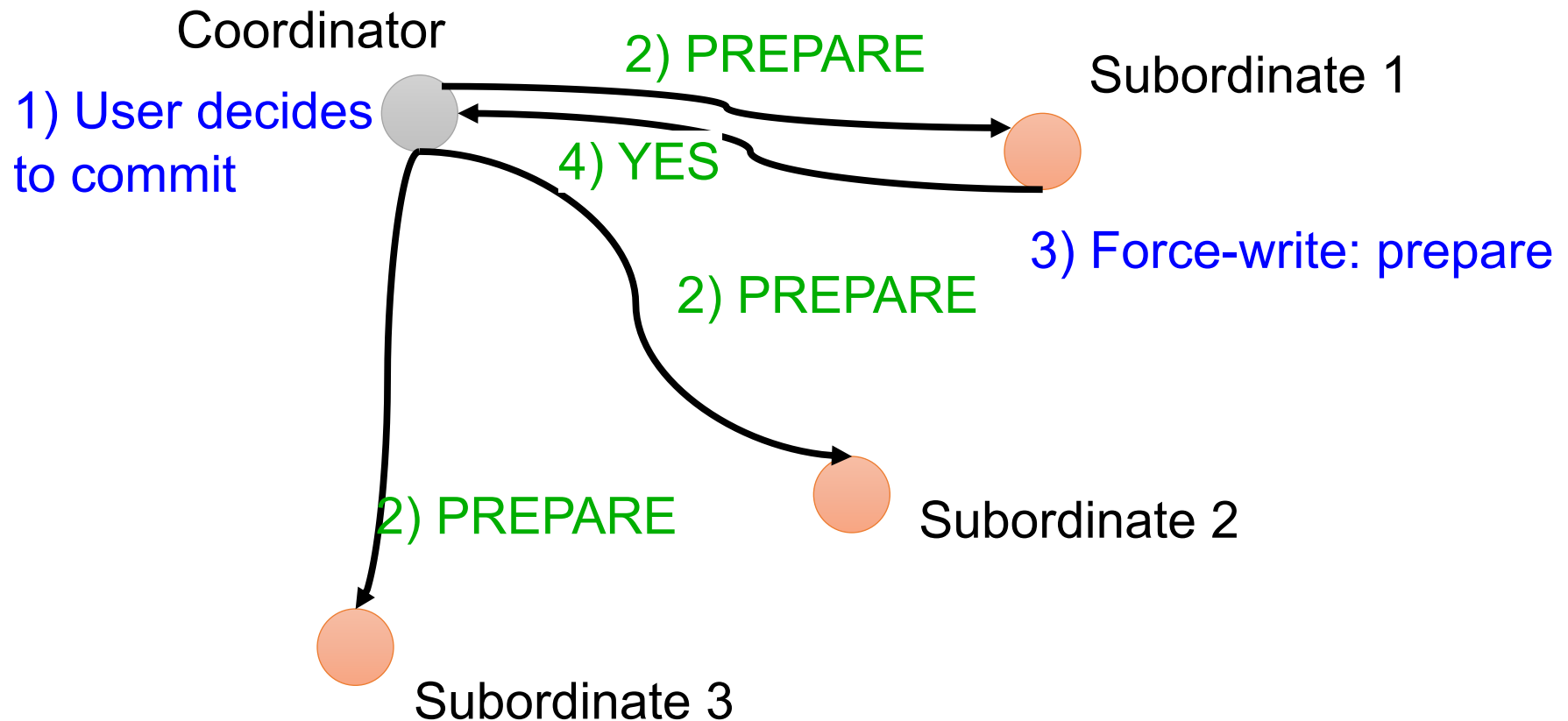
# 2PC: Phase 1, Prepare



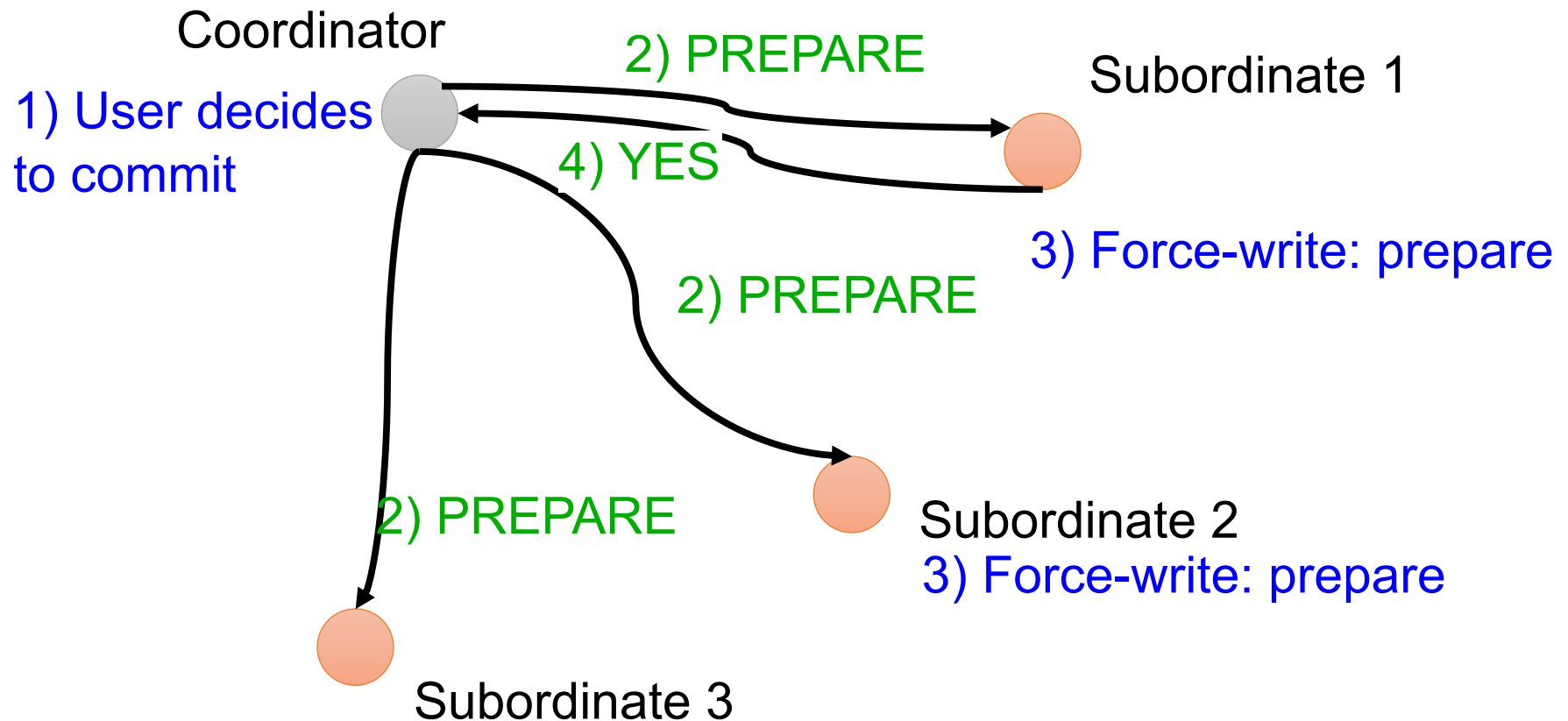
# 2PC: Phase 1, Prepare



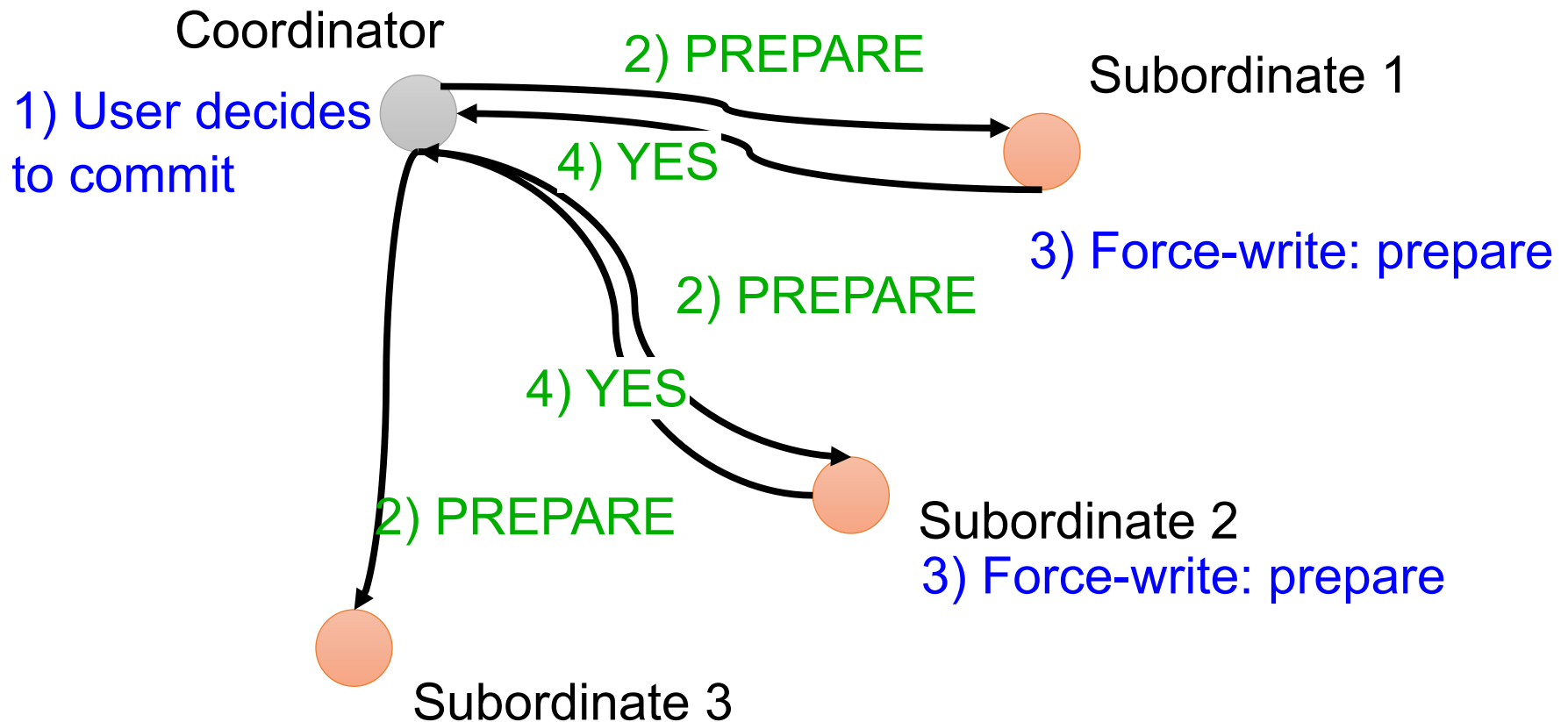
# 2PC: Phase 1, Prepare



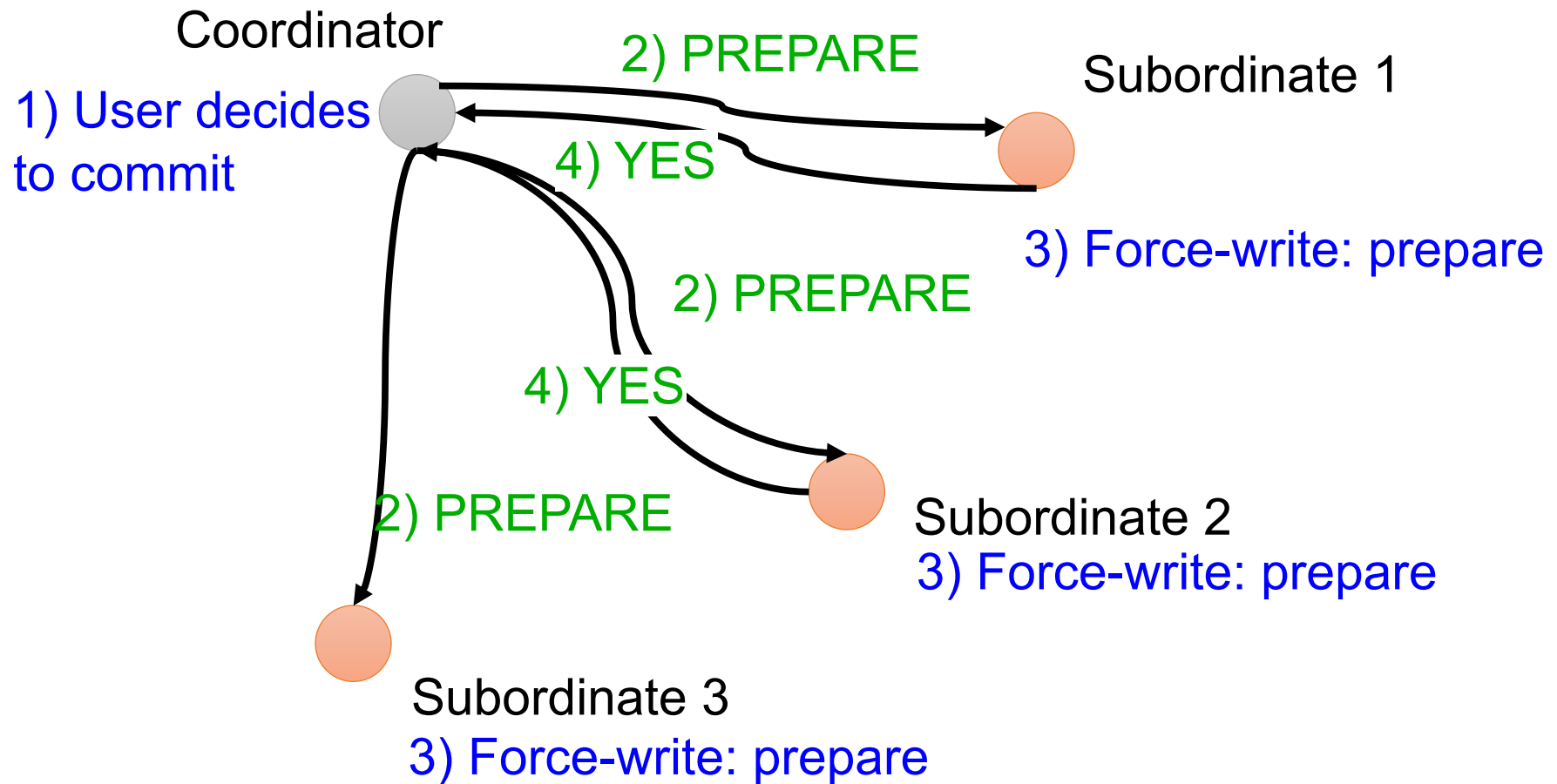
# 2PC: Phase 1, Prepare



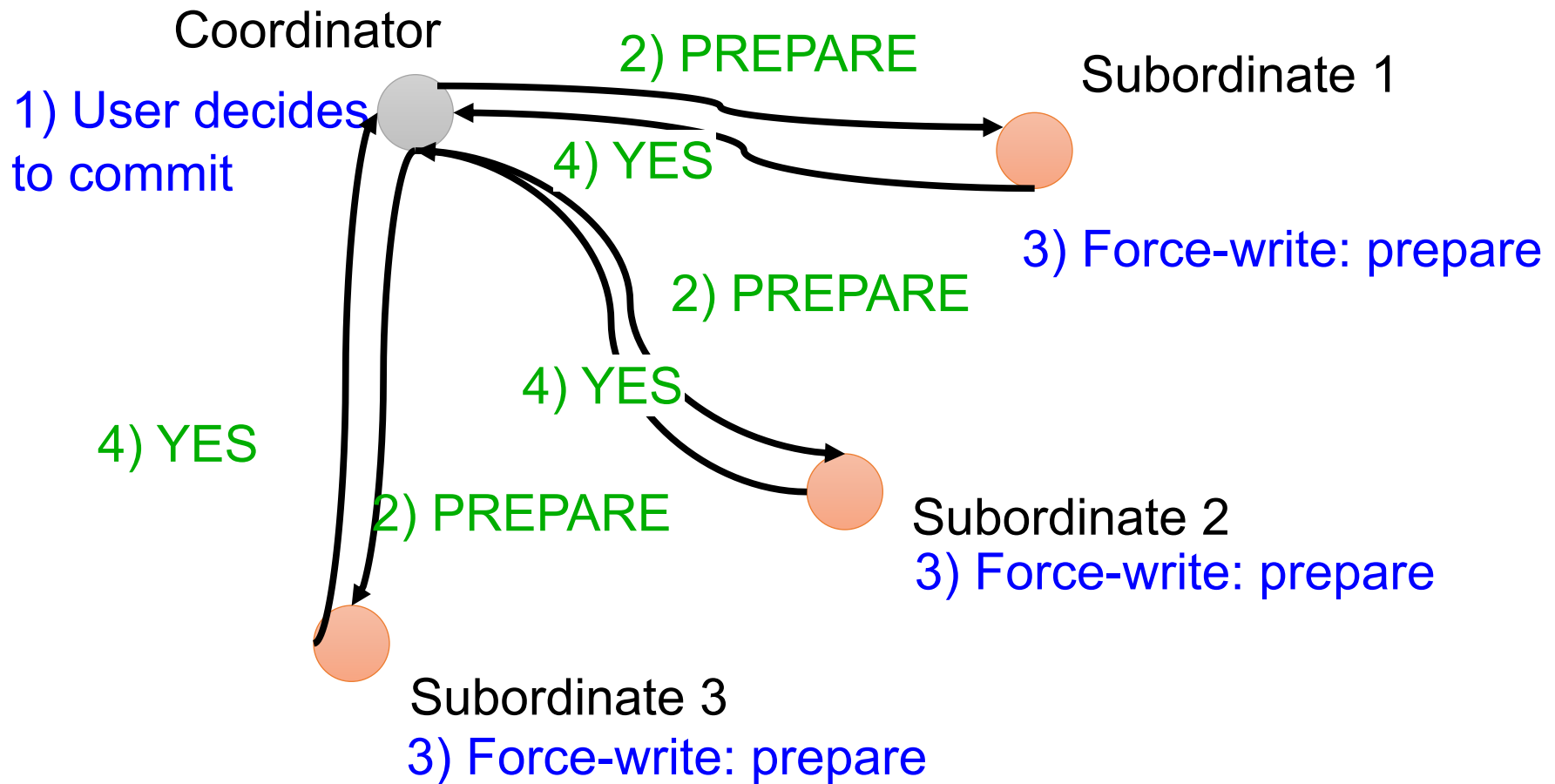
# 2PC: Phase 1, Prepare



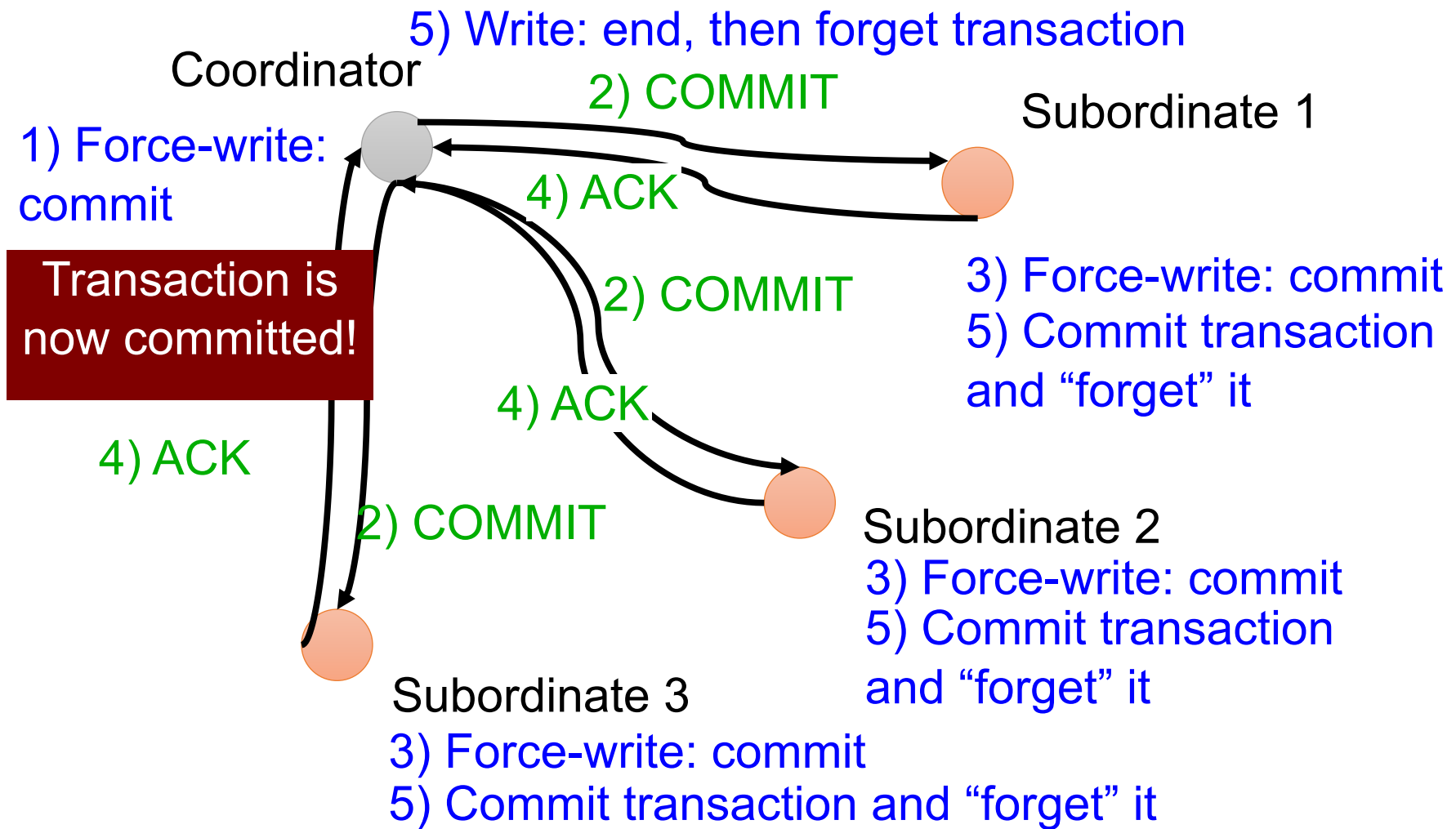
# 2PC: Phase 1, Prepare



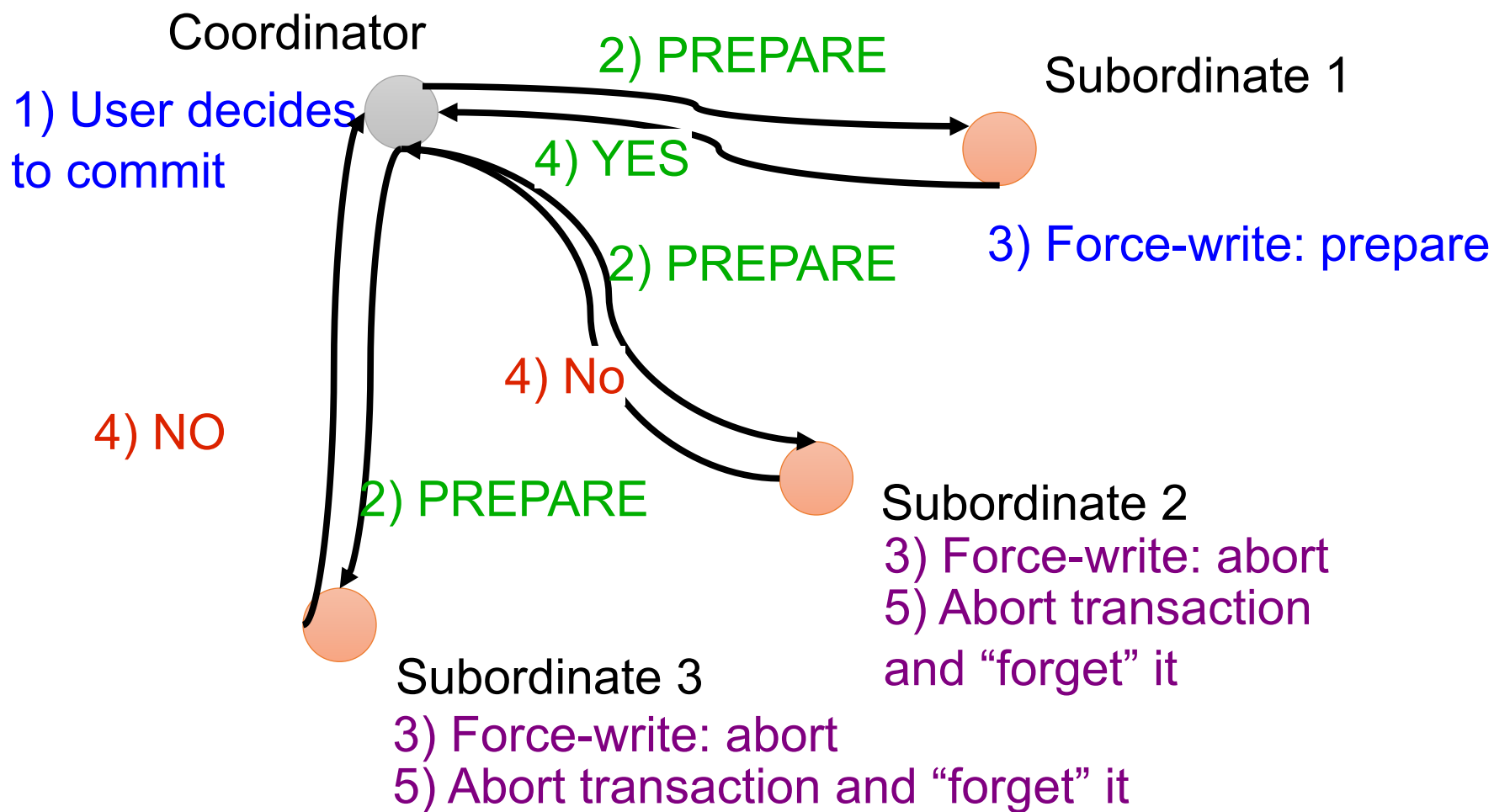
# 2PC: Phase 1, Prepare



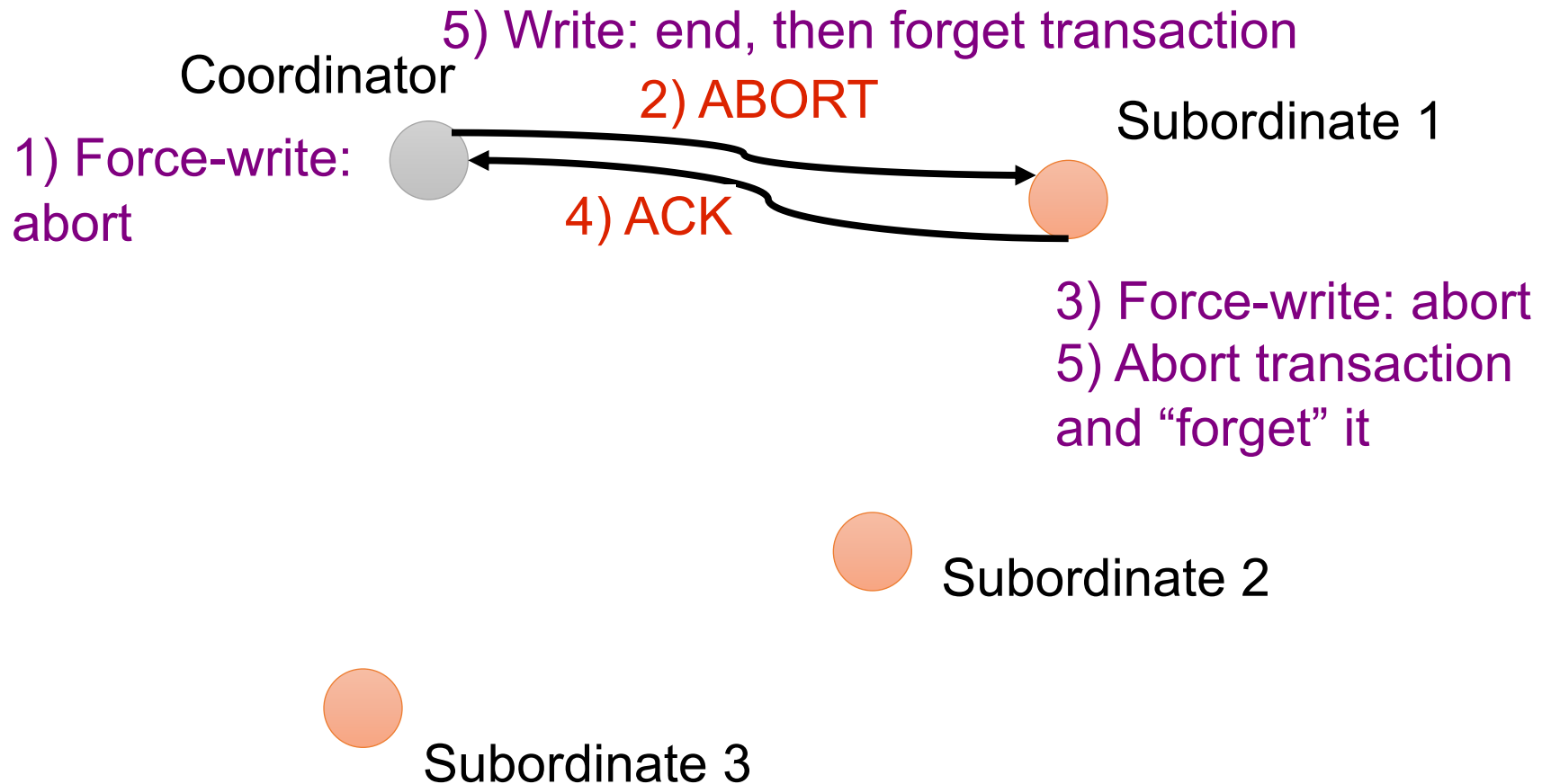
# 2PC: Phase 2, Commit



# 2PC with Abort – Phase 1



# 2PC with Abort – Phase 2

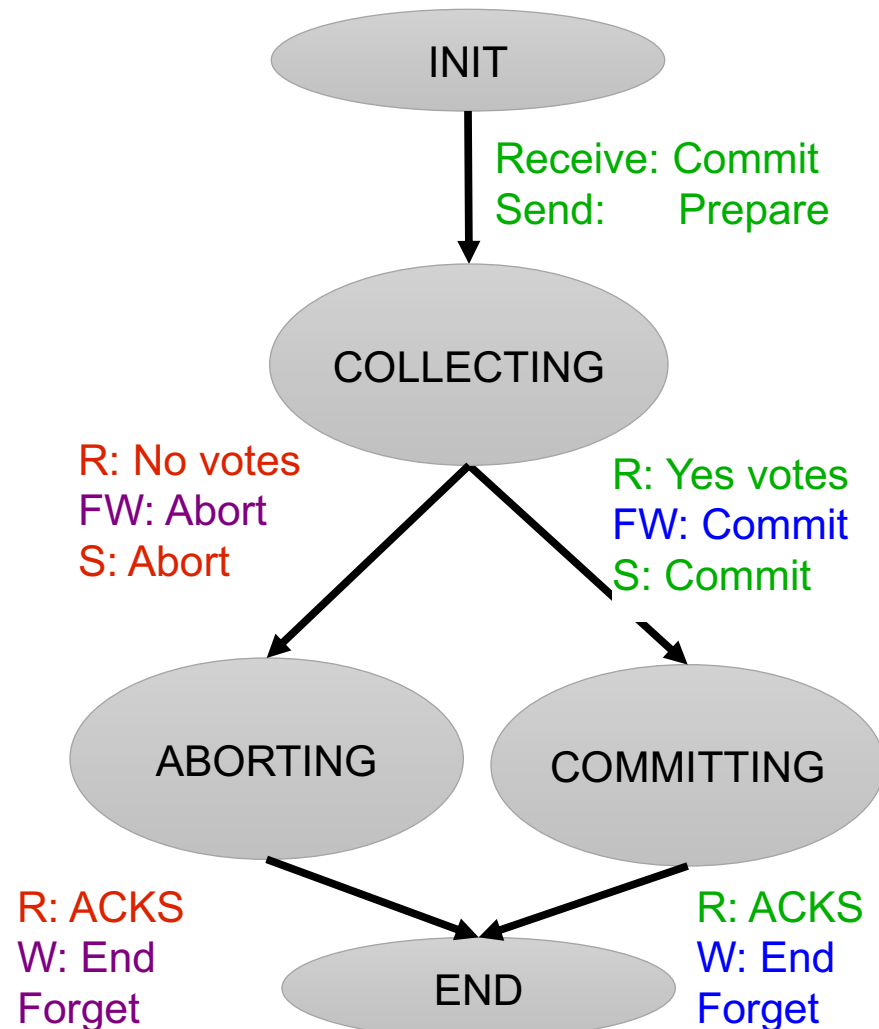


# Recap

- Phase 1, Prepare: collect votes
  - What if no response? Presume abort
- Phase 2, send decision commit/abort
  - Wait for ack then write END and forget

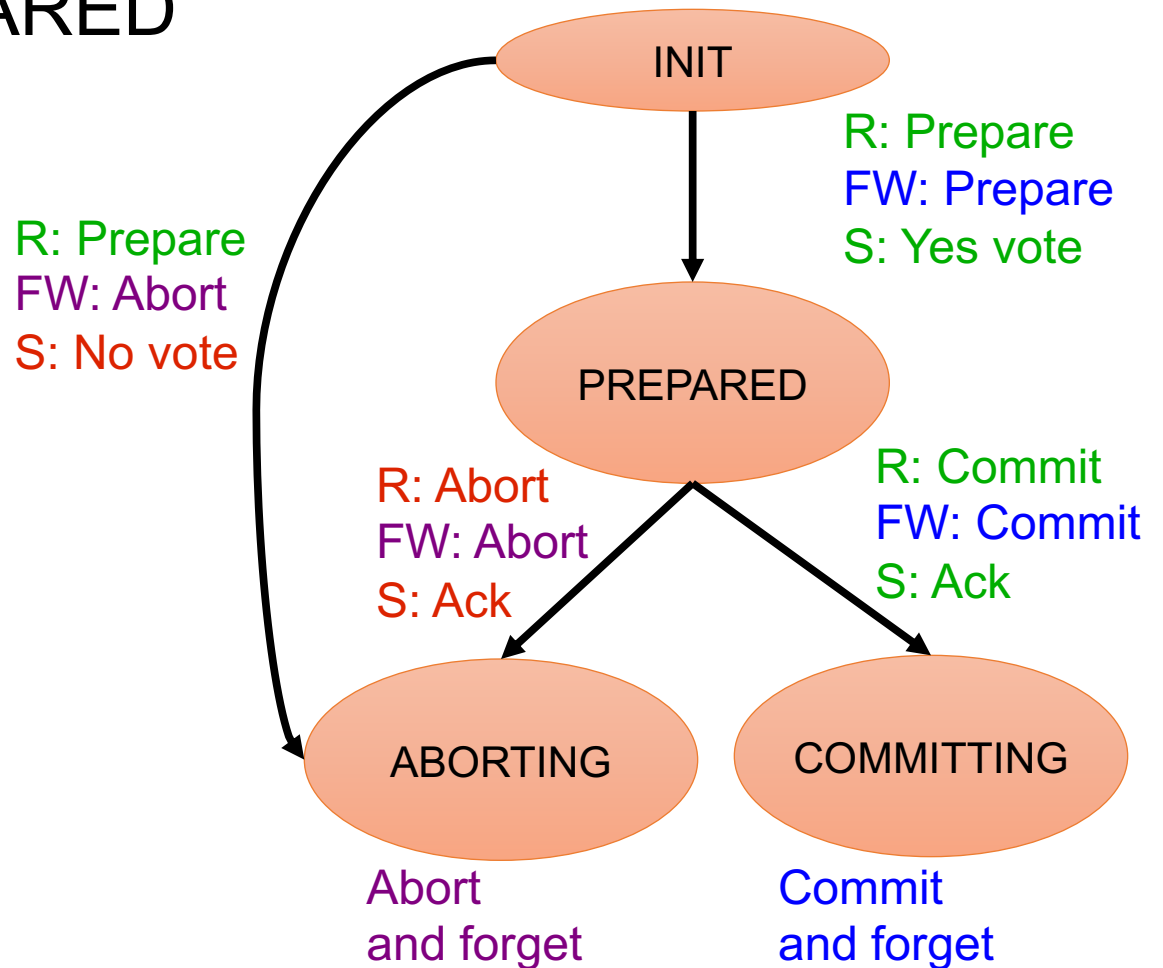
# Coordinator State Machine

- All states involve **waiting** for messages



# Subordinate State Machine

- INIT and PREPARED involve waiting



# Handling Site Failures

What to do if there is no response

- Approach 1: no site failure detection
  - Subordinate can only do retrying & **blocking**
- Approach 2: timeouts, since unilateral abort is ok
  - Subordinate: init state: can **timeout**;  
prepared state is still **blocking**
  - Coordinator: collecting state can **timeout**  
committing state is **blocking**
- **2PC is a blocking protocol**

# Recovery

A subordinate fails. During recovery:

- If the last entry in the log is <COMMIT T> then the transaction is committed: REDO

# Recovery

A subordinate fails. During recovery:

- If the last entry in the log is `<COMMIT T>` then the transaction is committed: REDO
- If the last entry in the log is `<ABORT T>`

# Recovery

A subordinate fails. During recovery:

- If the last entry in the log is `<COMMIT T>` then the transaction is committed: REDO
- If the last entry in the log is `<ABORT T>` then the transaction is aborted: UNDO

# Recovery

A subordinate fails. During recovery:

- If the last entry in the log is `<COMMIT T>` then the transaction is committed: `REDO`
- If the last entry in the log is `<ABORT T>` then the transaction is aborted: `UNDO`
- If no `COMMIT/ABORT/PREPARE` is found

# Recovery

A subordinate fails. During recovery:

- If the last entry in the log is `<COMMIT T>` then the transaction is committed: REDO
- If the last entry in the log is `<ABORT T>` then the transaction is aborted: UNDO
- If no COMMIT/ABORT/PREPARE is found, then presume ABORT (why is this OK?)

# Recovery

A subordinate fails. During recovery:

- If the last entry in the log is `<COMMIT T>` then the transaction is committed: REDO
- If the last entry in the log is `<ABORT T>` then the transaction is aborted: UNDO
- If no COMMIT/ABORT/PREPARE is found, then presume ABORT (why is this OK?)
- If the last entry is `<PREPARE T>` then it's hard:

# Recovery

A subordinate fails. During recovery:

- If the last entry in the log is `<COMMIT T>` then the transaction is committed: REDO
- If the last entry in the log is `<ABORT T>` then the transaction is aborted: UNDO
- If no COMMIT/ABORT/PREPARE is found, then presume ABORT (why is this OK?)
- If the last entry is `<PREPARE T>` then it's hard: must re-contact coordinator to find out whether ABORT or COMMIT

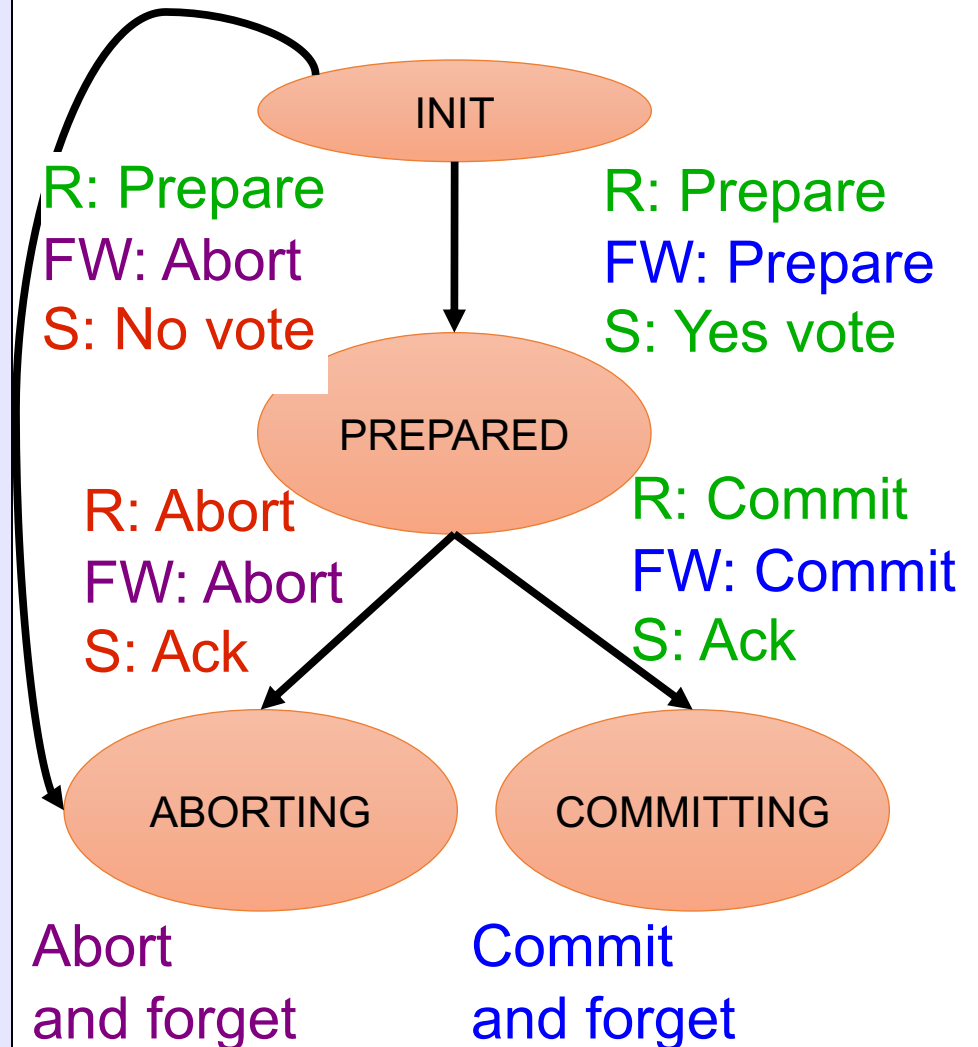
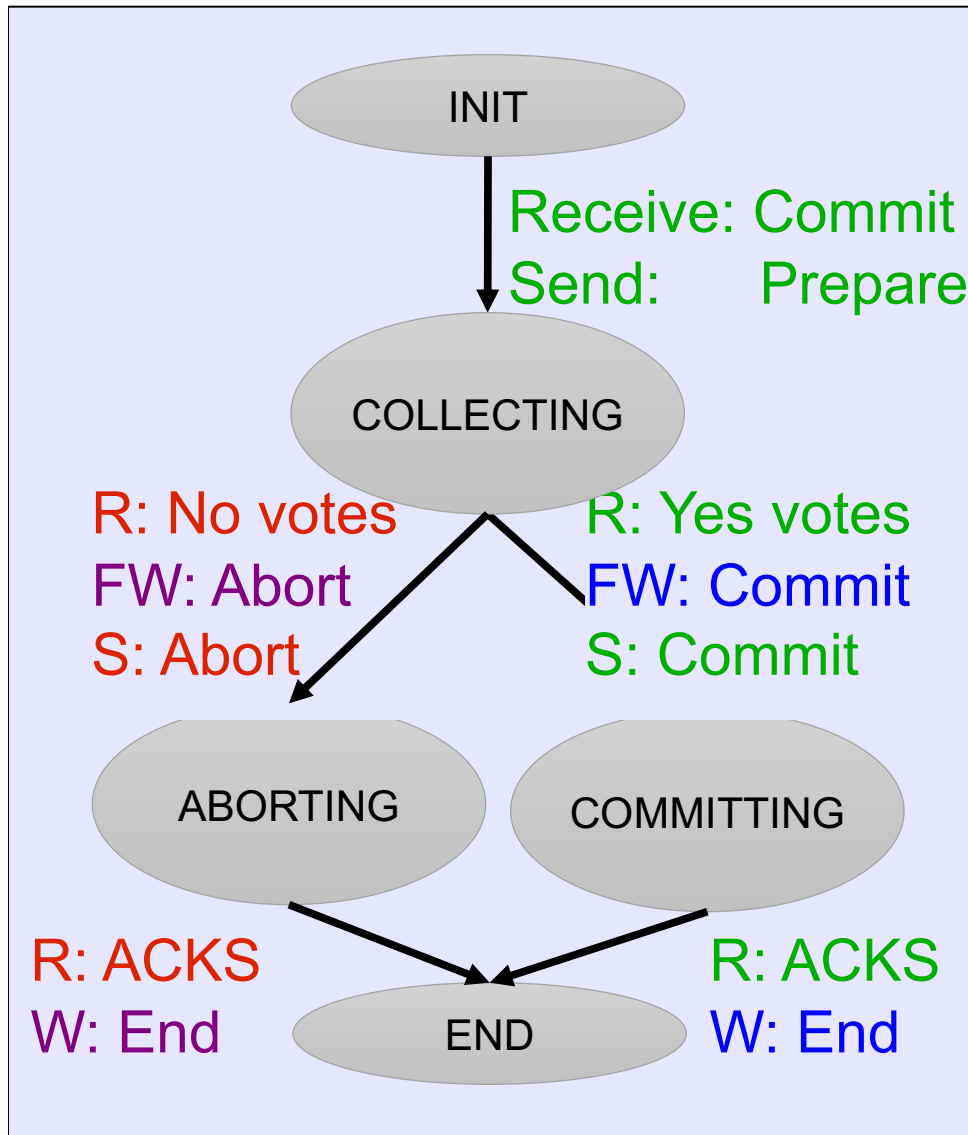
# Observations

- Coordinator keeps transaction in transactions table until it receives all acks
  - To ensure subordinates know to commit or abort
  - So acks enable coordinator to “forget” about transaction
- After crash, if recovery process finds no log records for a transaction, the transaction is presumed to have aborted
- Read-only subtransactions: no changes ever need to be undone nor redone

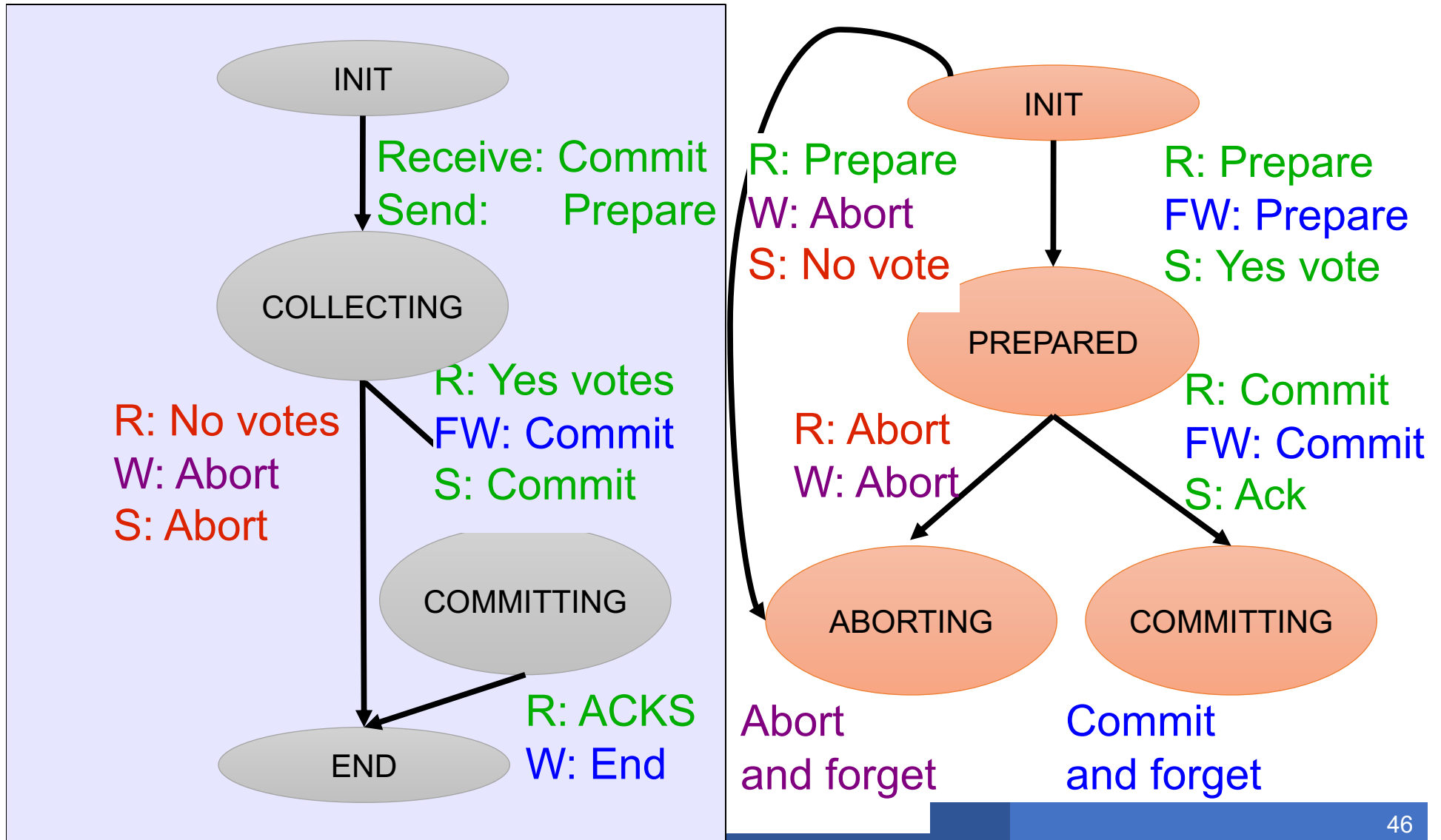
# Presumed Abort Protocol

- Optimization goals
  - Fewer messages and fewer force-writes
- Principle
  - If nothing known about a transaction, assume ABORT
- Aborting transactions need no force-writing
- Avoid log records for read-only transactions
  - Reply with a READ vote instead of YES vote

# 2PC State Machines (repeat)



# Presumed Abort State Machines



# Summary: Two-Phase Commit Protocol

- One coordinator and many subordinates
  - Phase 1: prepare
    - All subordinates must flush tail of write-ahead log to disk before ack
    - Must ensure that if coordinator decides to commit, they can commit!
  - Phase 2: commit or abort
  - Log records for 2PC include transaction and coordinator ids
  - Coordinator also logs ids of all subordinates
- Principle
  - Whenever a process makes a decision: vote yes/no or commit/abort
  - Or whenever a subordinate wants to respond to a message: ack
  - **First force-write a log record** (to make sure it survives a failure)
  - **Only then send message about decision**
- “Forget” completed transactions at the very end
  - Once synchronized, or transaction has committed or aborted, all nodes can stop logging any more information about that transaction

# Discussion

- Data replication: simple case of distributed TXN: ensure that all replicas performed the update
  - But 2PC is slow: waiting for the slowest link
  - Major shortcoming: need reliable coordinator
  - Paxos: gives up the coordinator, even slower...
- 
- NoSQL: give up strong consistency (i.e. ACID)
  - Mostly for data replication: “eventual consistency”
  - Programming nightmare: how to write a TXN?