

# Distributed OLTP Databases (Part II)



Lecture #23



Database Systems  
15-445/15-645  
Fall 2018

AP

Andy Pavlo  
Computer Science  
Carnegie Mellon Univ.

# LAST CLASS

---

## **System Architectures**

→ Shared-Memory, Shared-Disk, Shared-Nothing

## **Partitioning/Sharding**

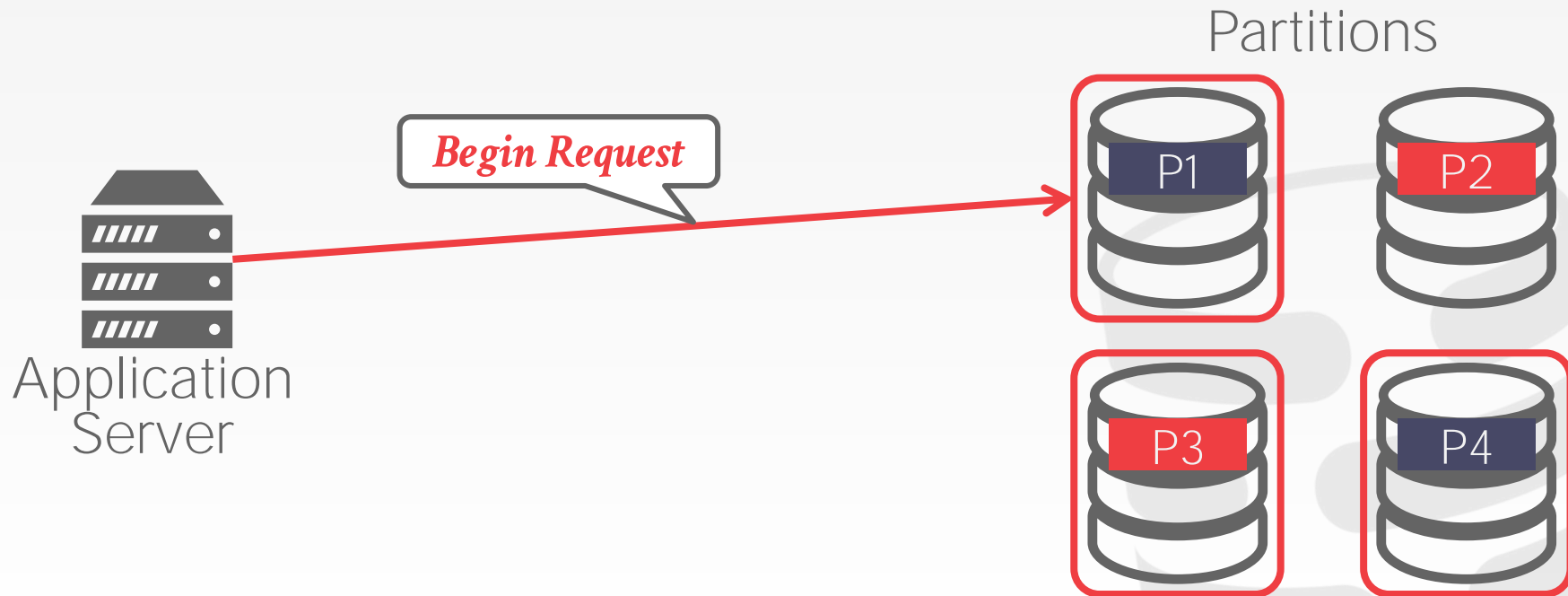
→ Hash, Range, Round Robin

## **Transaction Coordination**

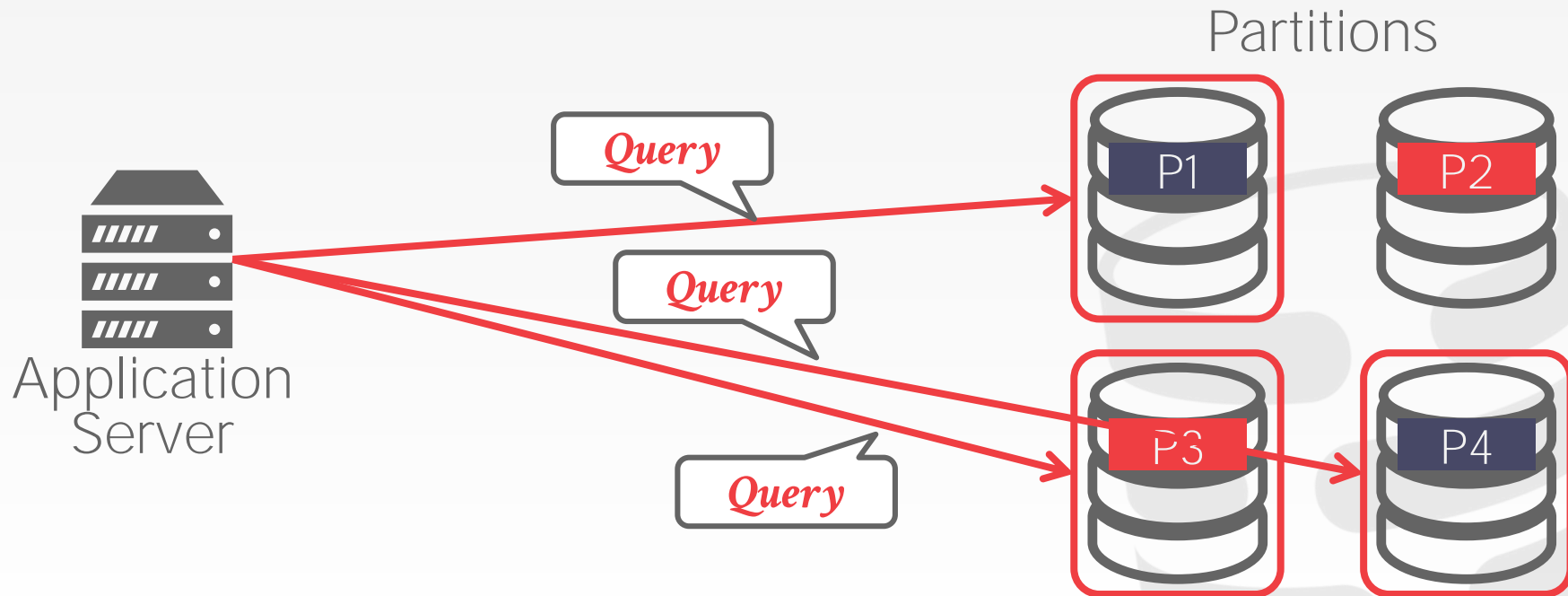
→ Centralized vs. Decentralized



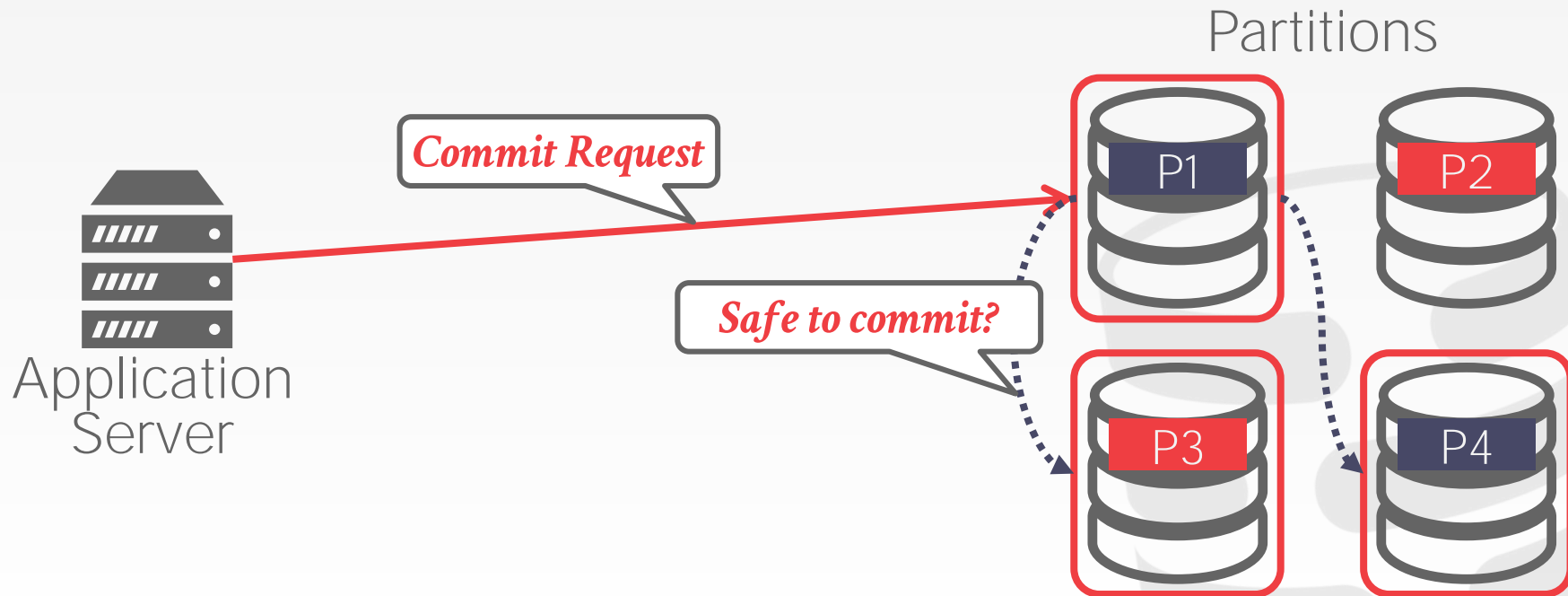
# DECENTRALIZED COORDINATOR



# DECENTRALIZED COORDINATOR



# DECENTRALIZED COORDINATOR



# OBSERVATION

---

We have not discussed how to ensure that all nodes agree to commit a txn and then to make sure it does commit if we decide that it should.

- What happens if a node fails?
- What happens if our messages show up late?
- What happens if we don't wait for every node to agree?

# TODAY'S AGENDA

---

Atomic Commit Protocols

Replication

Consistency Issues (CAP)

Federated Databases



# ATOMIC COMMIT PROTOCOL

---

When a multi-node txn finishes, the DBMS needs to ask all of the nodes involved whether it is safe to commit.

Examples:

- Two-Phase Commit
- Three-Phase Commit (not used)
- Paxos
- Raft
- ZAB (Apache Zookeeper)
- Viewstamped Replication

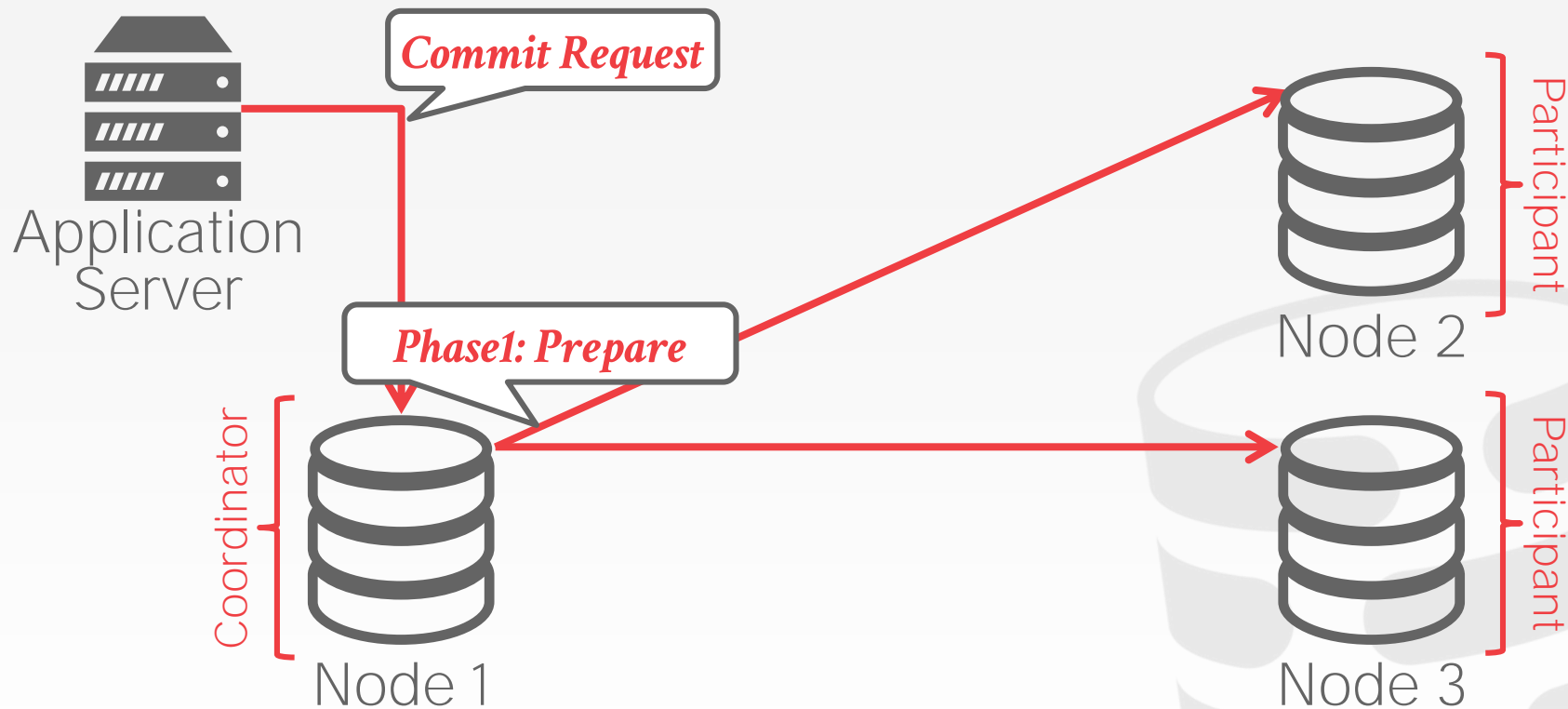




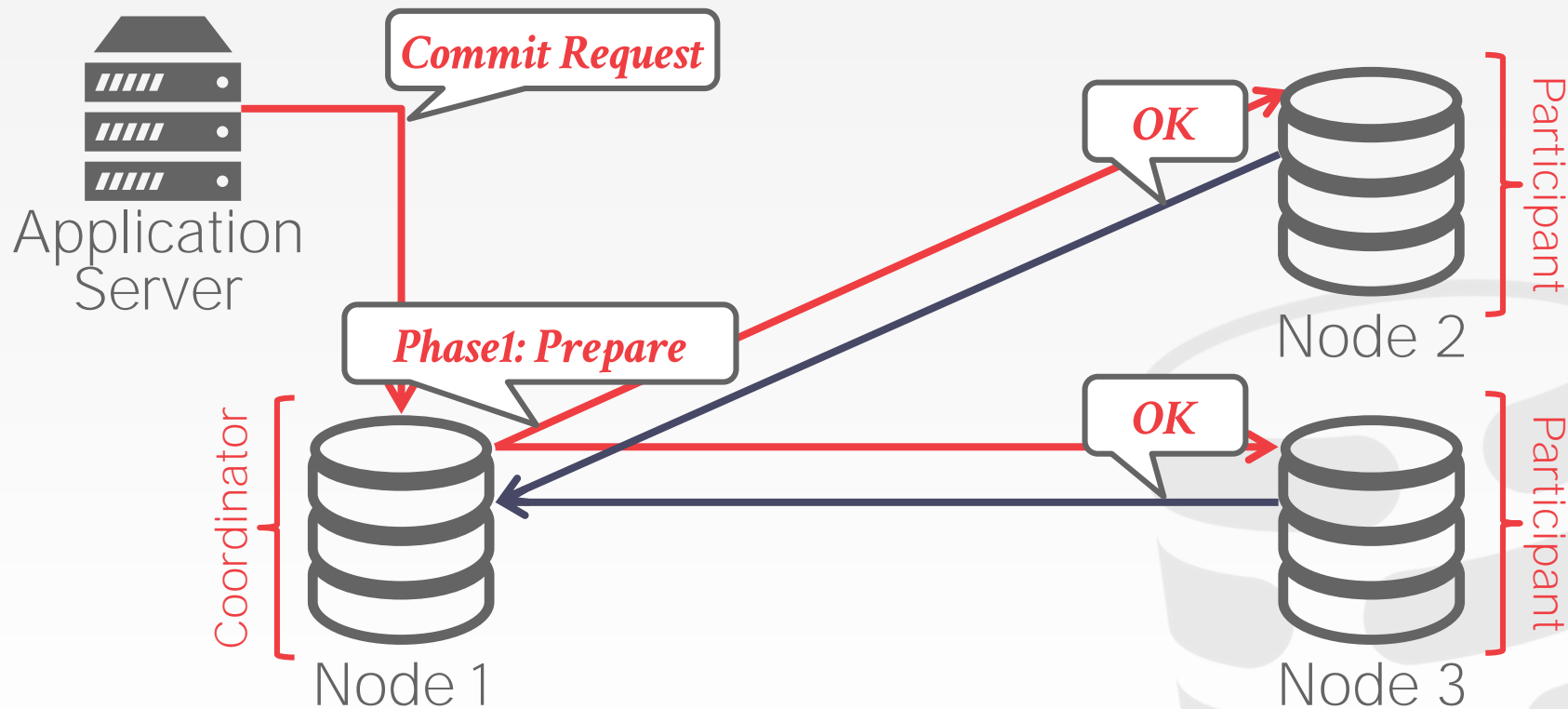
# TWO-PHASE COMMIT (SUCCESS)



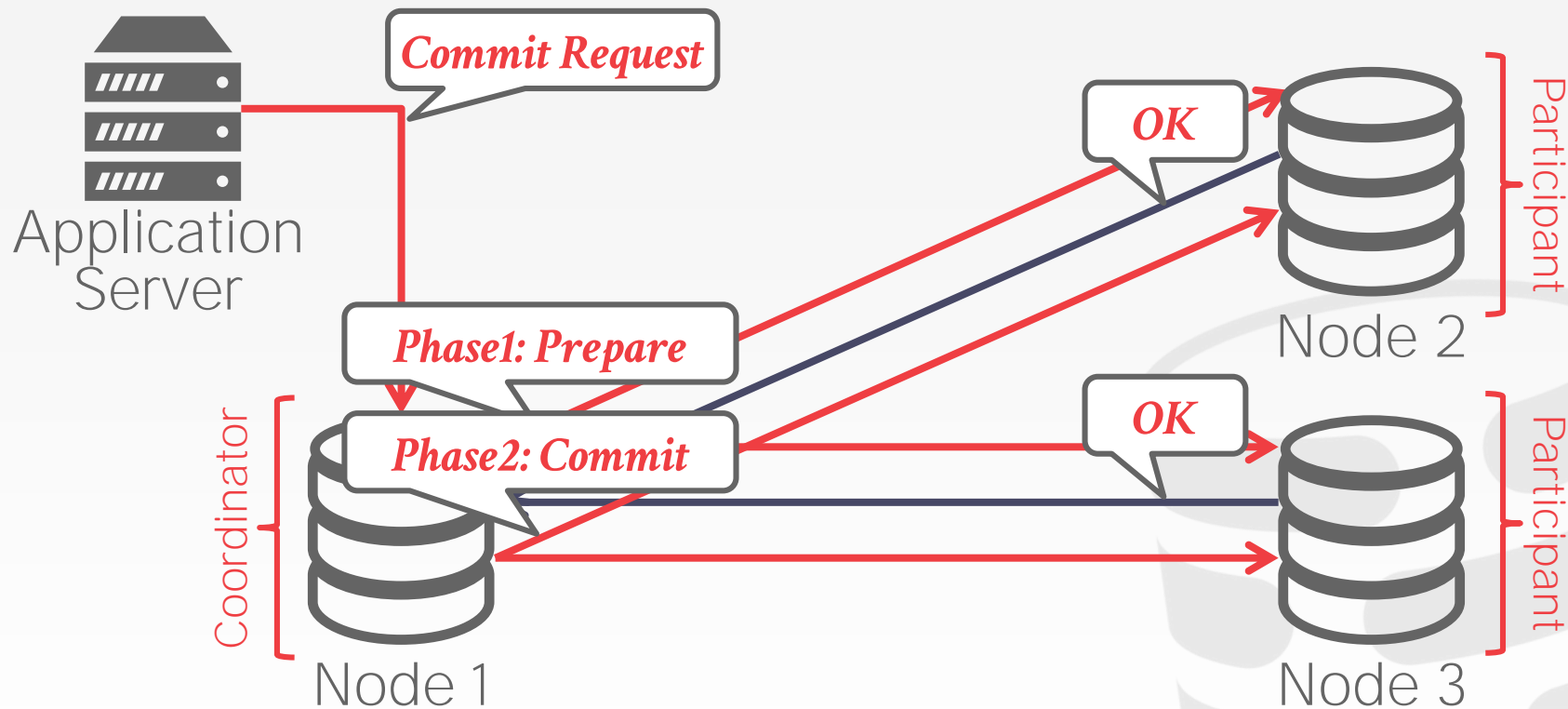
# TWO-PHASE COMMIT (SUCCESS)



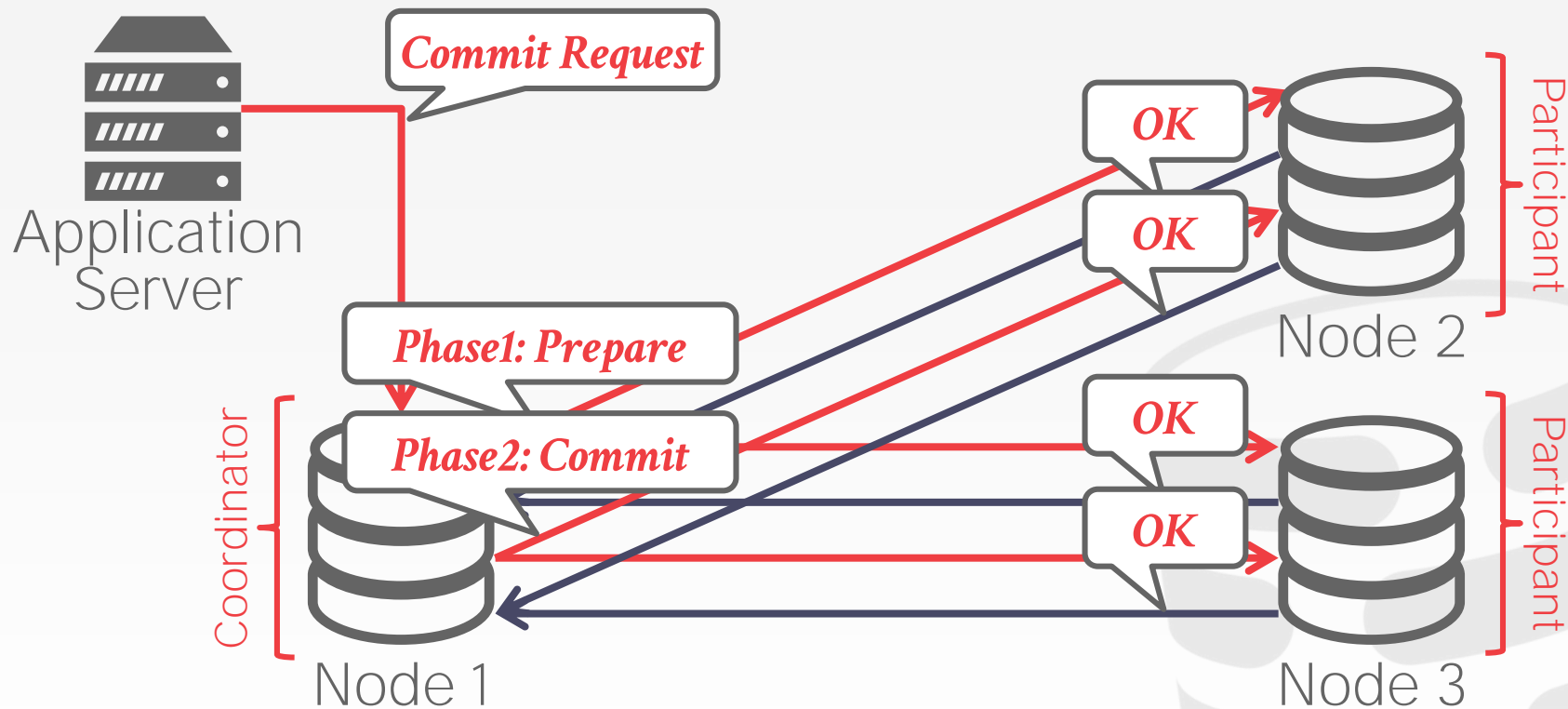
# TWO-PHASE COMMIT (SUCCESS)



# TWO-PHASE COMMIT (SUCCESS)



# TWO-PHASE COMMIT (SUCCESS)



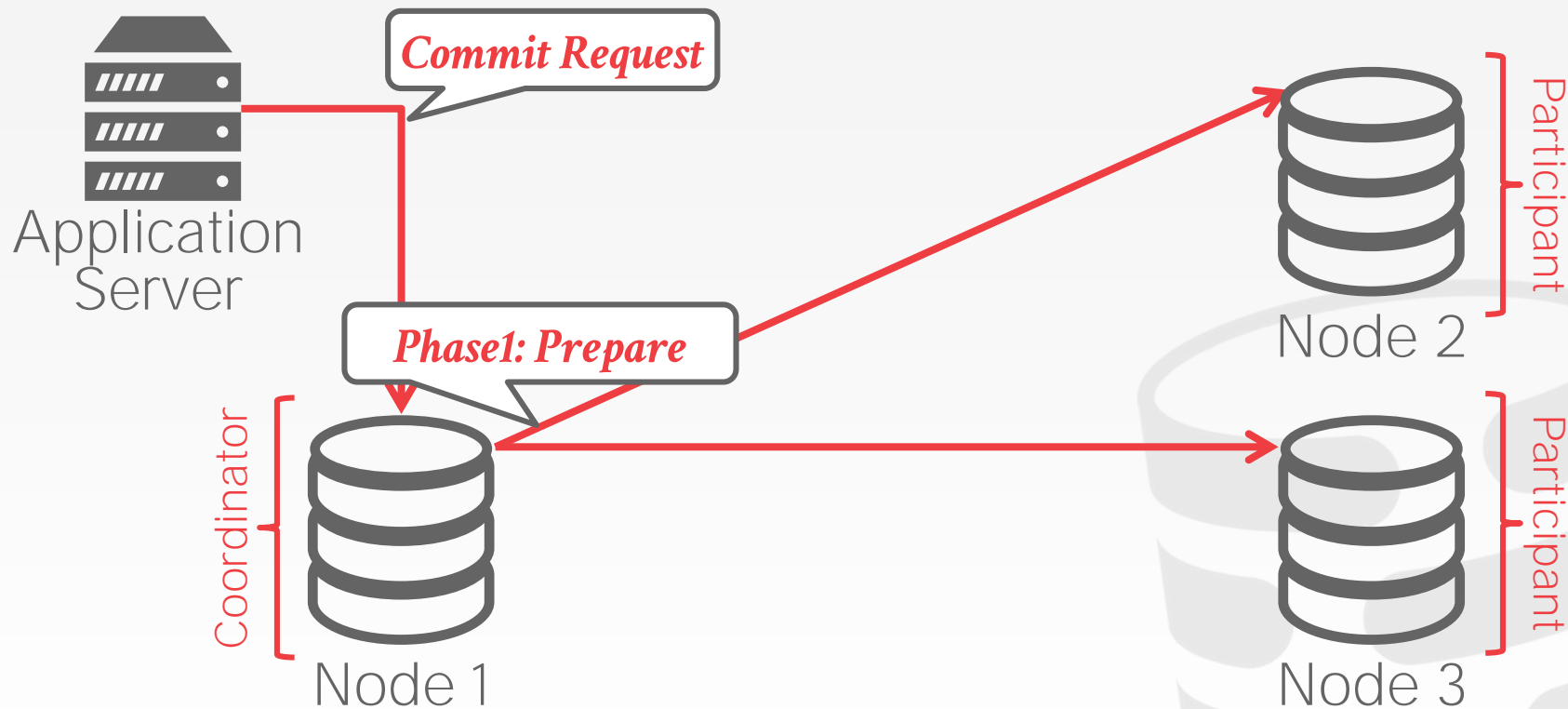
# TWO-PHASE COMMIT (SUCCESS)



# TWO-PHASE COMMIT (ABORT)

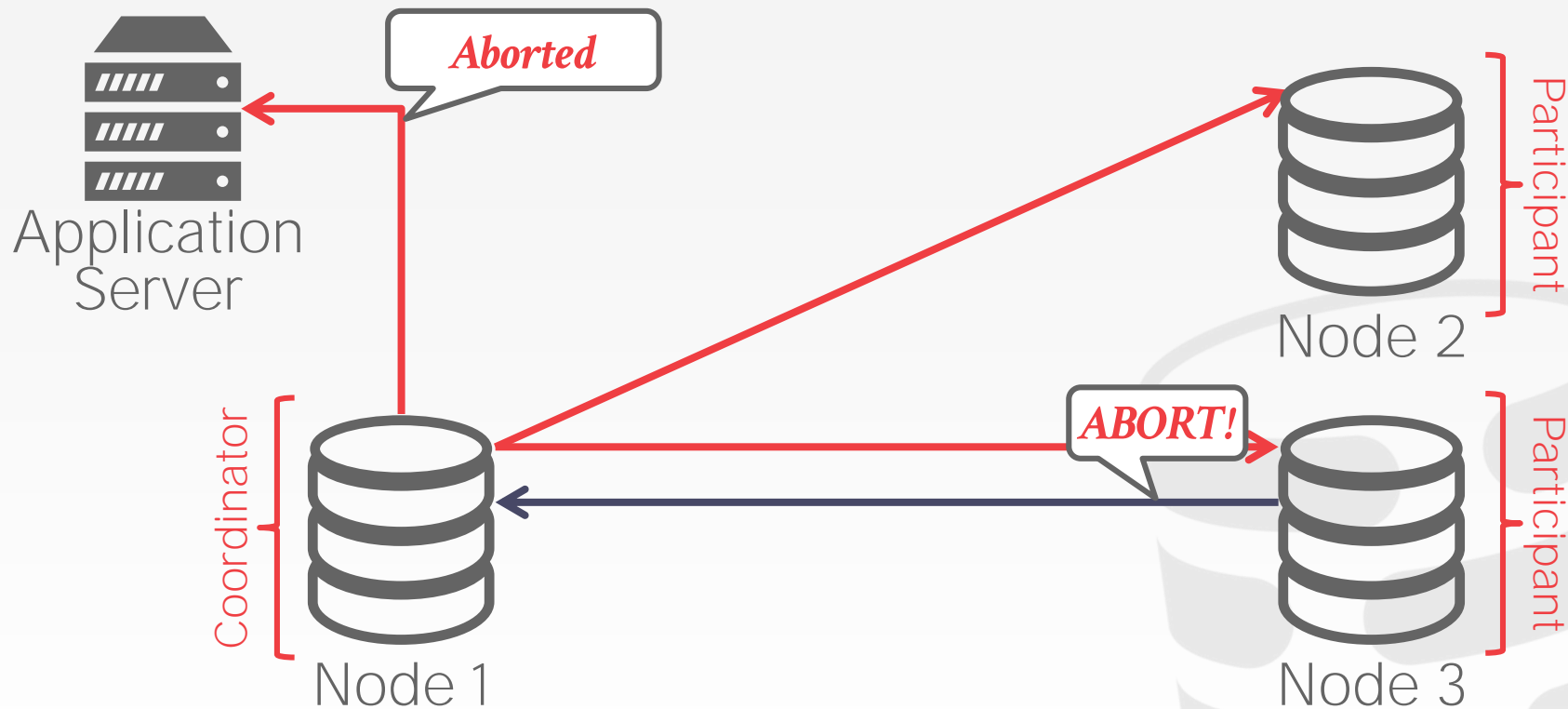


# TWO-PHASE COMMIT (ABORT)

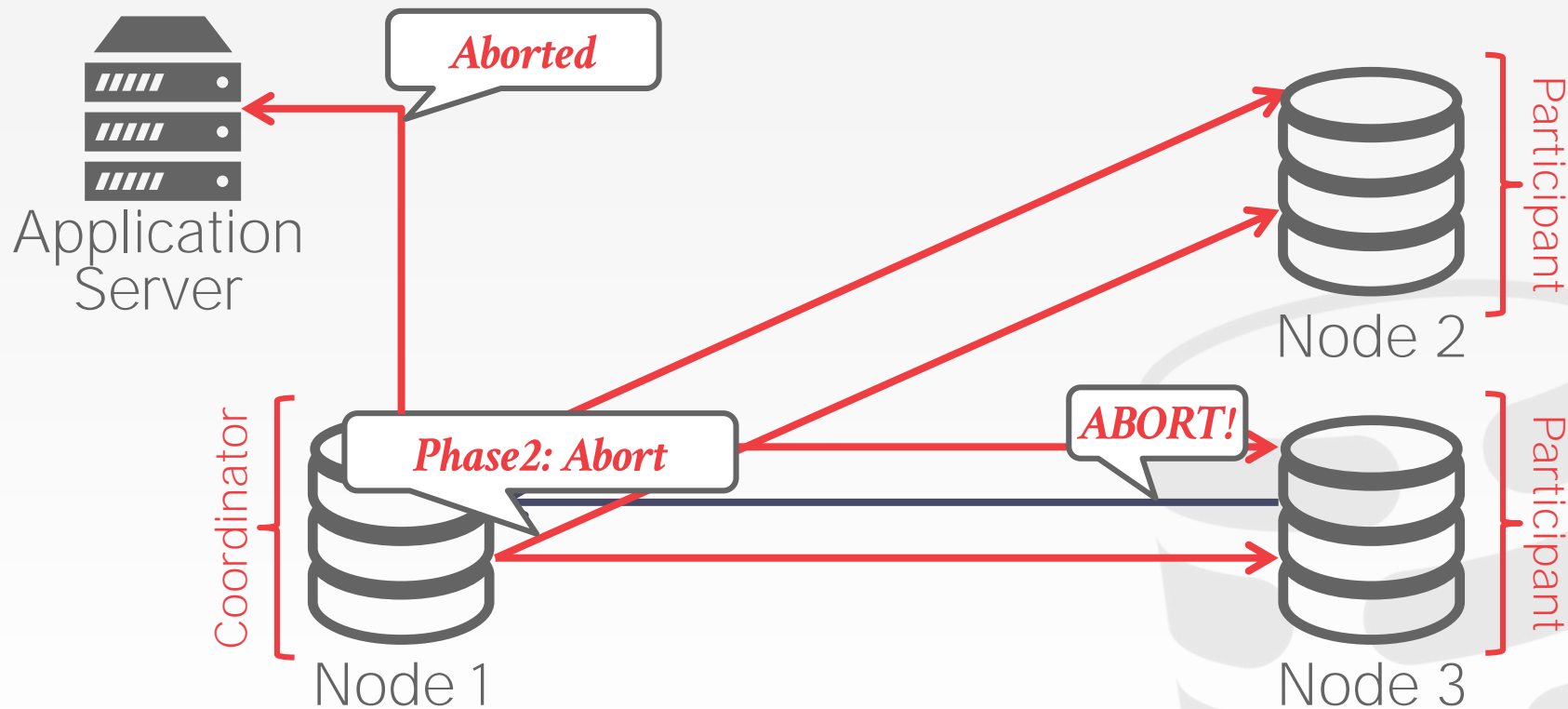




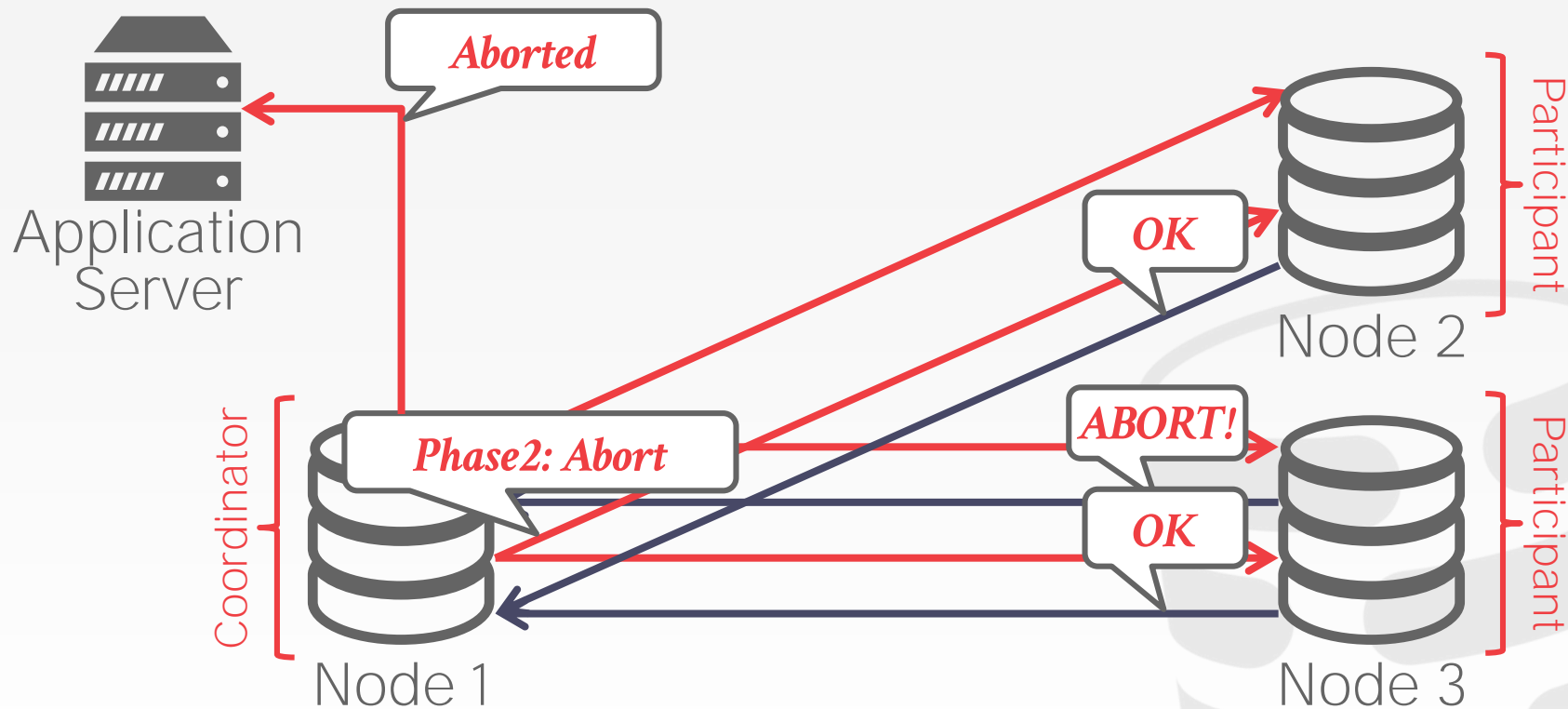
## TWO-PHASE COMMIT (ABORT)



## TWO-PHASE COMMIT (ABORT)



## TWO-PHASE COMMIT (ABORT)



# 2PC OPTIMIZATIONS

---

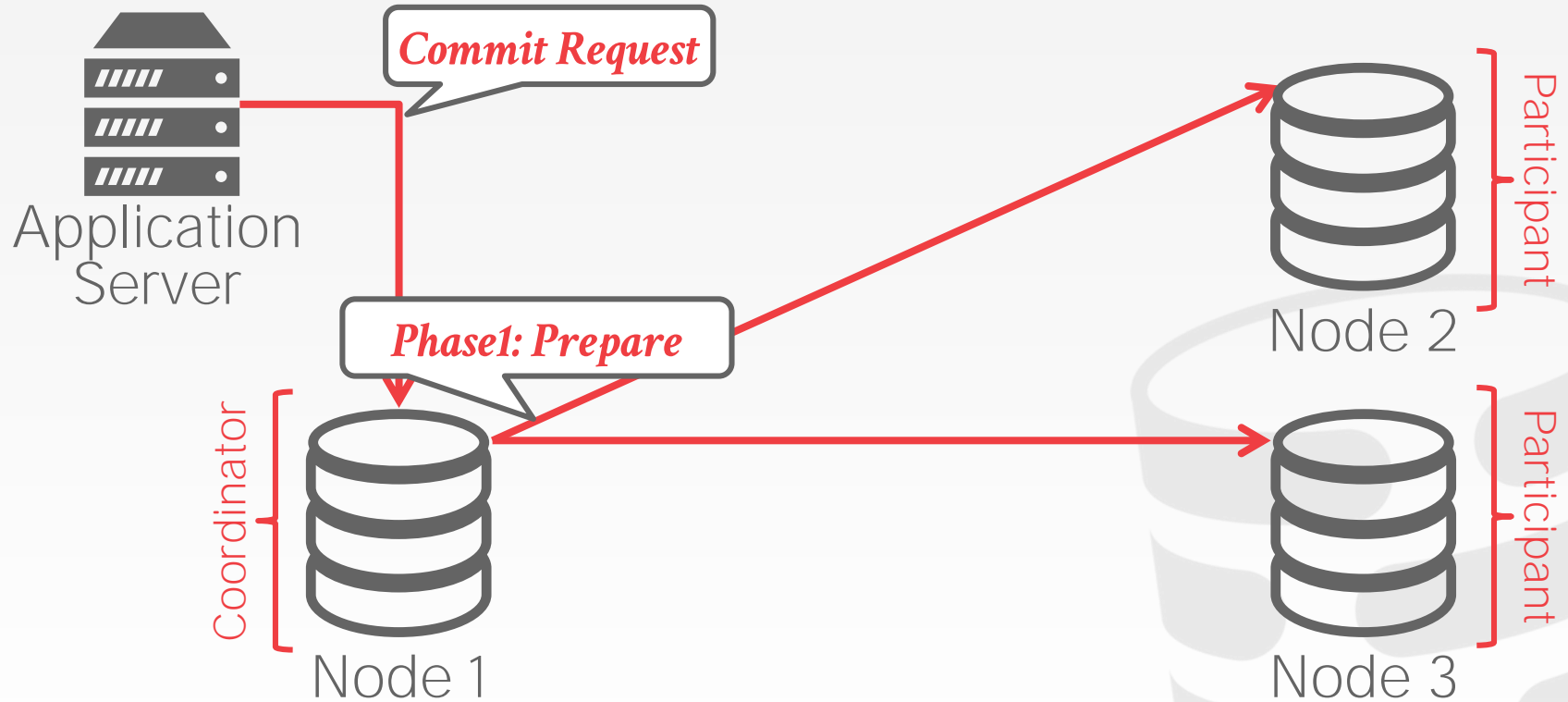
## Early Prepare Voting

- If you send a query to a remote node that you know will be the last one you execute there, then that node will also return their vote for the prepare phase with the query result.

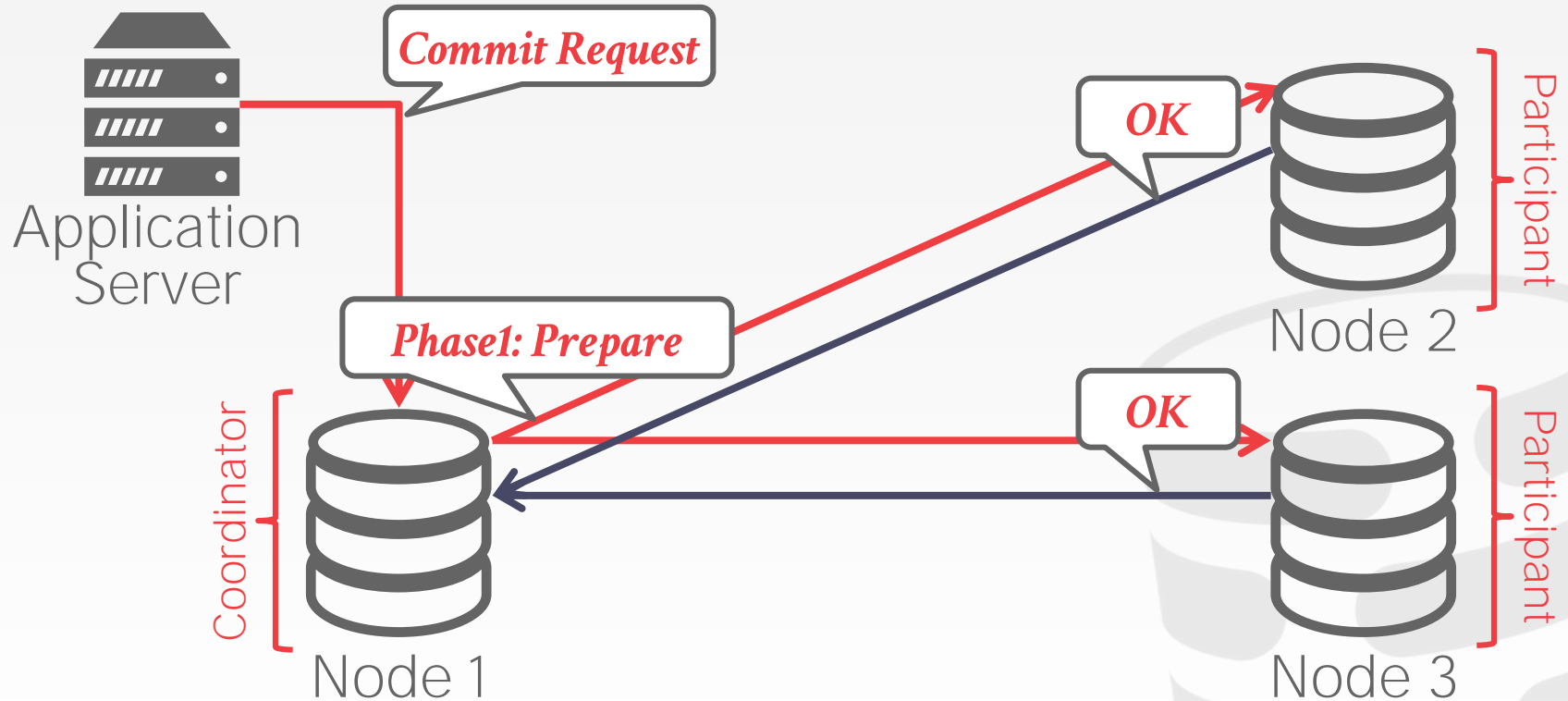
## Early Acknowledgement After Prepare

- If all nodes vote to commit a txn, the coordinator can send the client an acknowledgement that their txn was successful before the commit phase finishes.

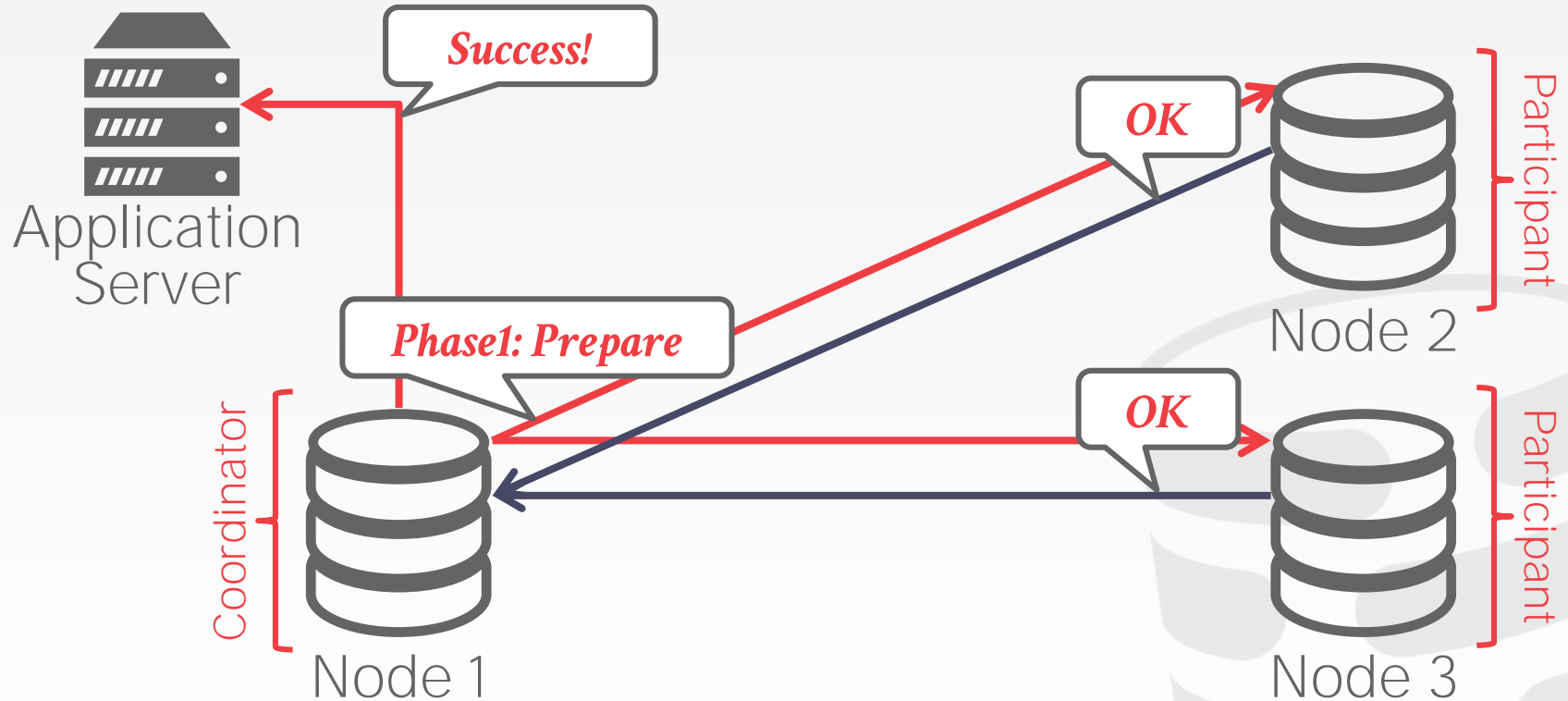
# EARLY ACKNOWLEDGEMENT



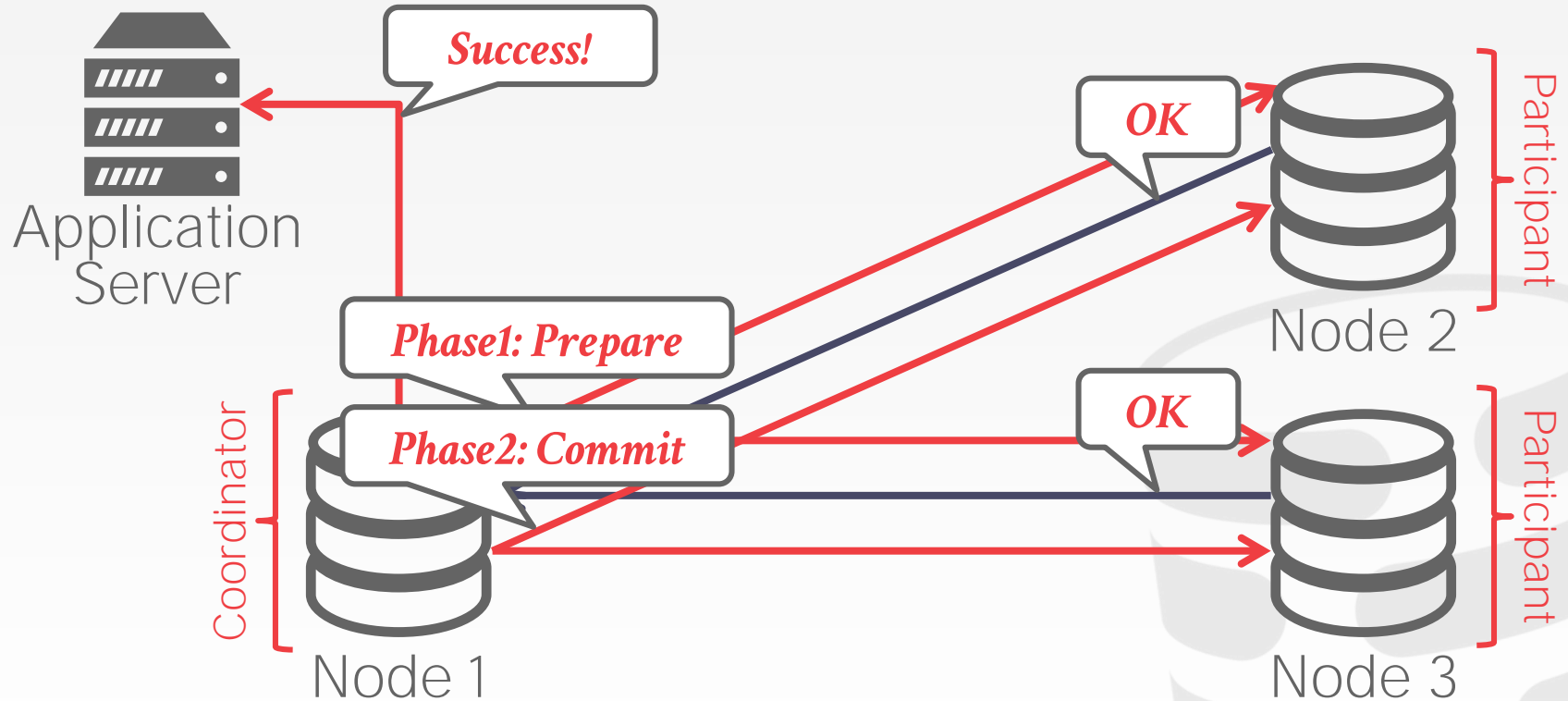
# EARLY ACKNOWLEDGEMENT



# EARLY ACKNOWLEDGEMENT

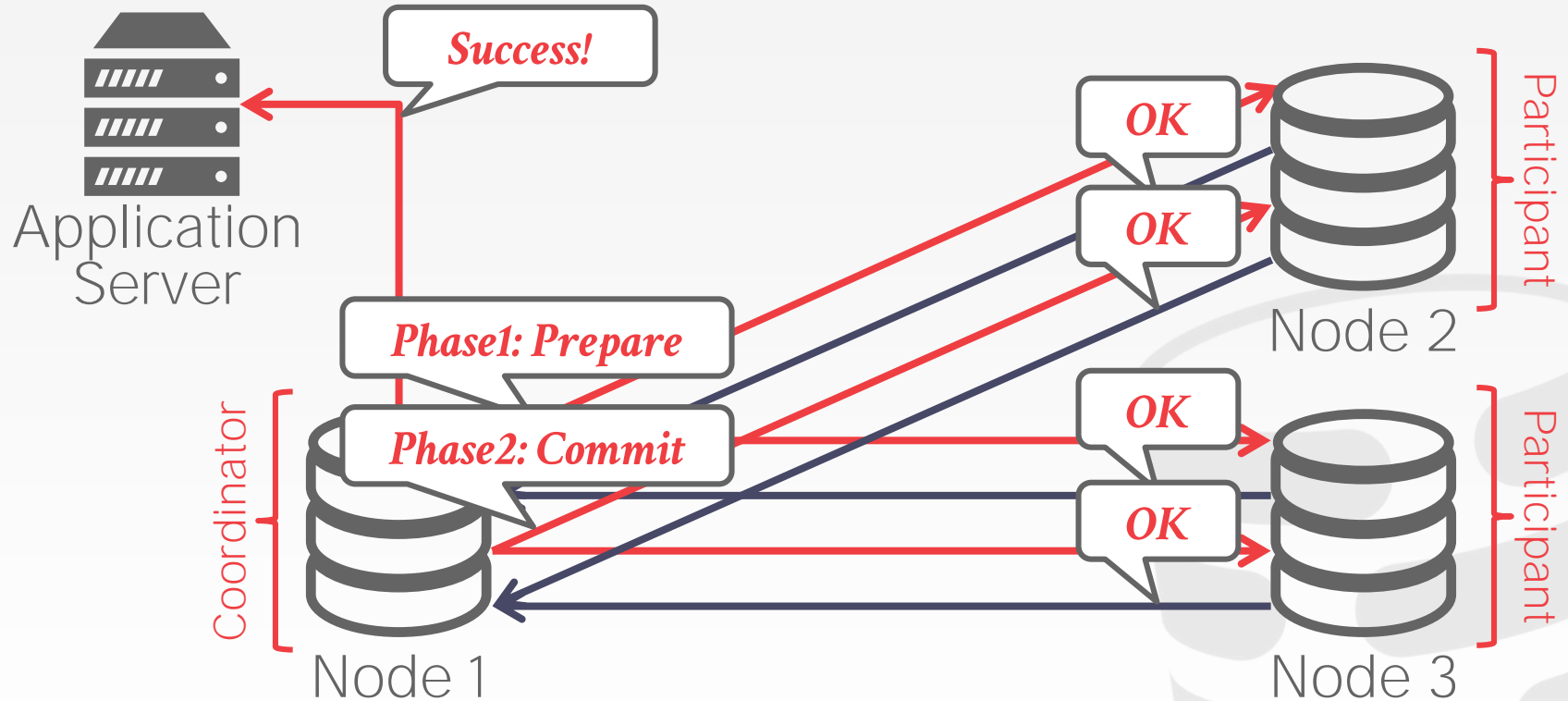


# EARLY ACKNOWLEDGEMENT





# EARLY ACKNOWLEDGEMENT



# TWO-PHASE COMMIT

---

Each node has to record the outcome of each phase in a stable storage log.

## **What happens if coordinator crashes?**

→ Participants have to decide what to do.

## **What happens if participant crashes?**

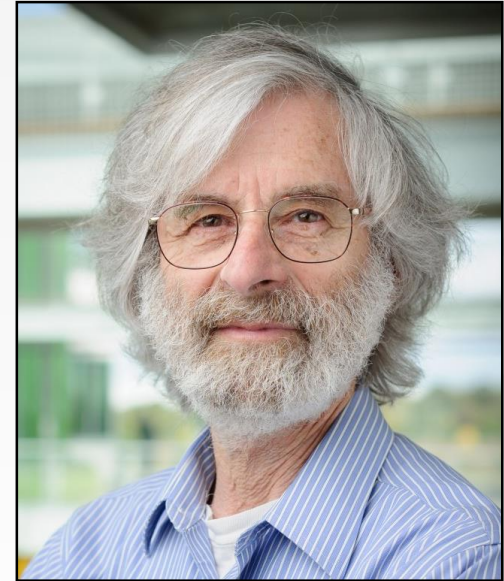
→ Coordinator assumes that it responded with an abort if it hasn't sent an acknowledgement yet.

# PAXOS

---

Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed.

Does not block if a majority of participants are available and has provably minimal message delays in the best case.



Lamport

# PAXOS

Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed.

Does not block if a majority of participants are available and has provably minimal message delays in the best case.

## The Part-Time Parliament

LESLIE LAMPORT  
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxos parliament's protocol provides a new way of implementing the state-machine approach to the design of distributed systems.

Categories and Subject Descriptors: C2.4 [Computer-Communications Networks]: Distributed Systems—Network operating systems; D4.5 [Operating Systems]: Reliability—Fault tolerance; J.1 [Administrative Data Processing]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

This submission was recently discovered behind a filing cabinet in the TOCS editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate, even though the obscure ancient Paxos civilization he describes is of little interest to most computer scientists; its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxos made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxos Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lamport [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

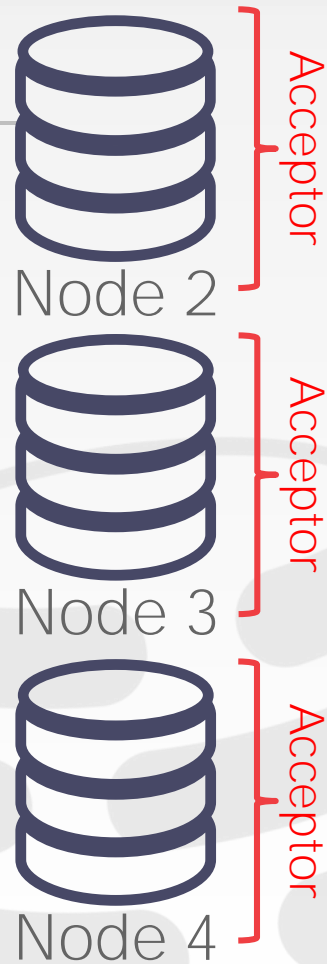
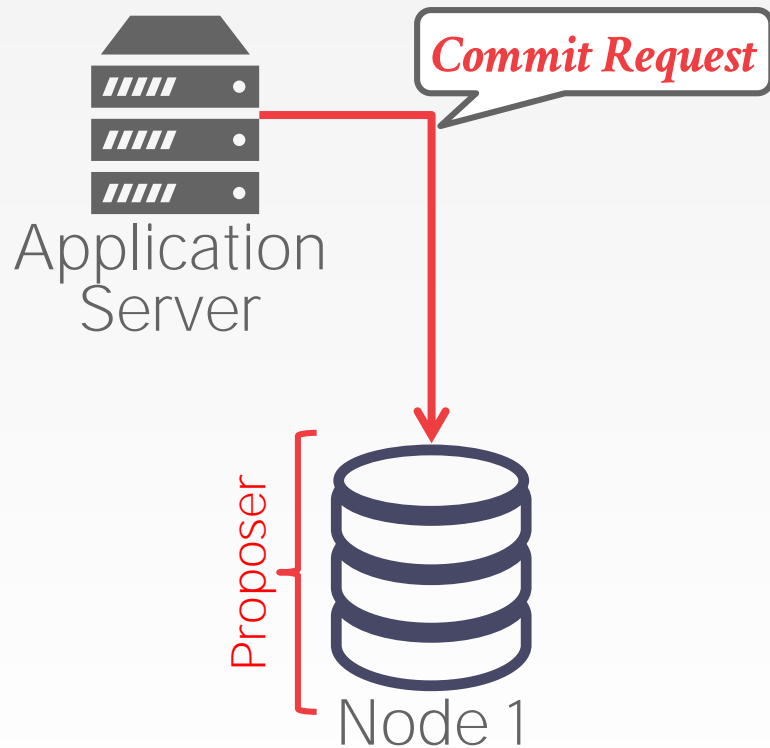
Keith Marzullo  
University of California, San Diego

Authors' address: Systems Research Center, Digital Equipment Corporation, 130 Lytton Avenue, Palo Alto, CA 94301.

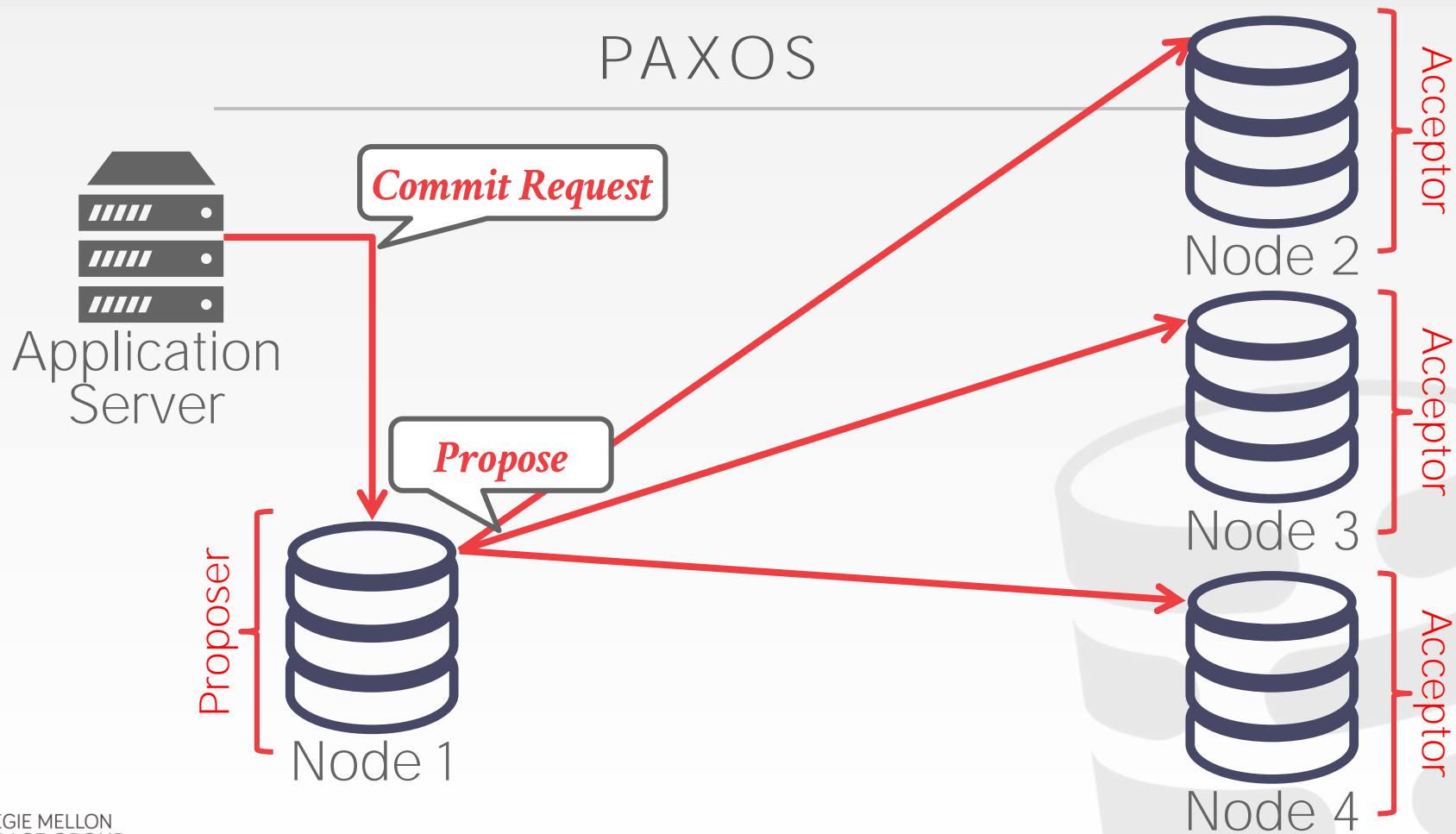
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, to republish, requires a fee and/or specific permission.

© 1998 ACM 0000-0000/98/0000-0000 \$00.00

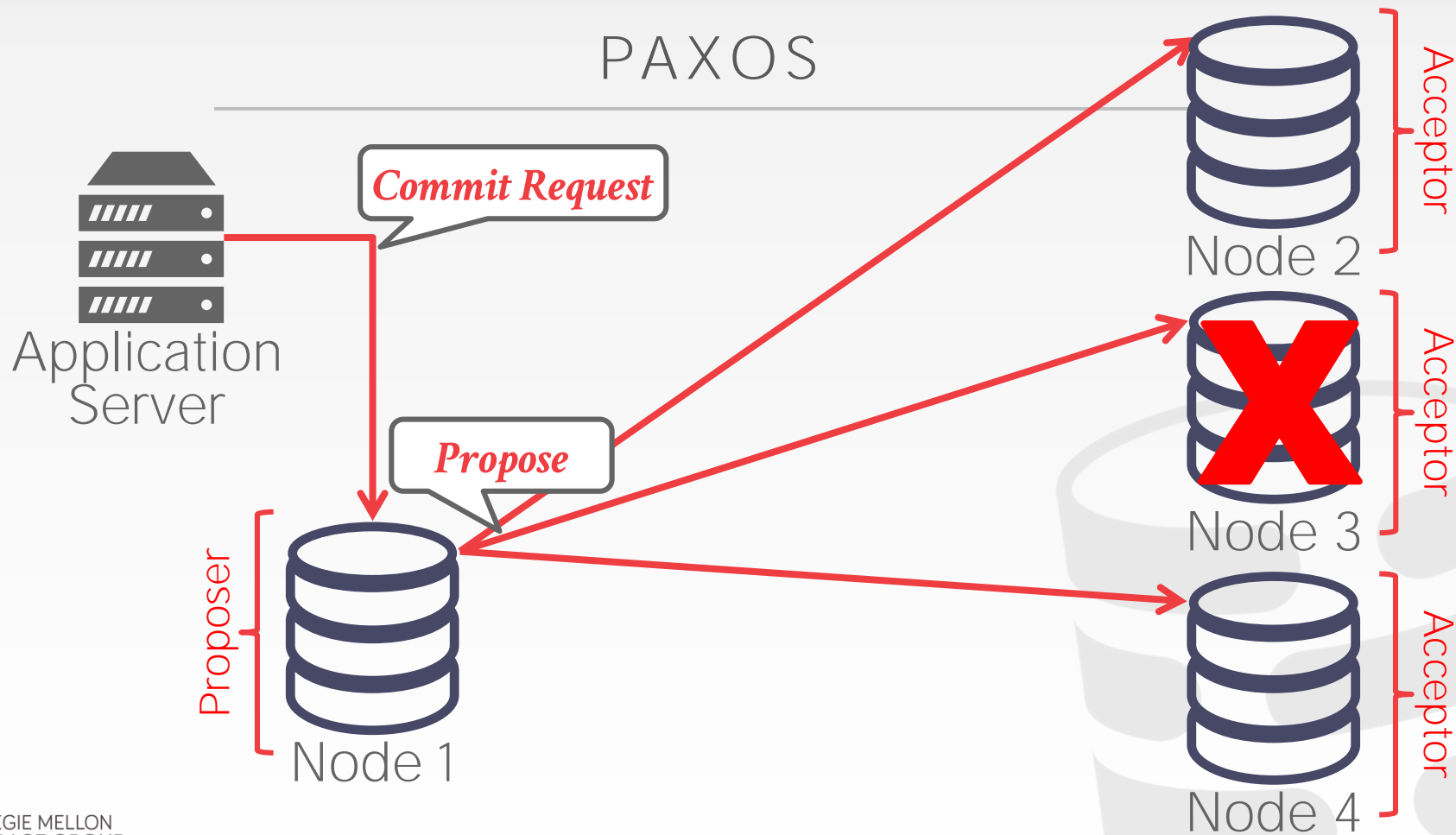
# PAXOS

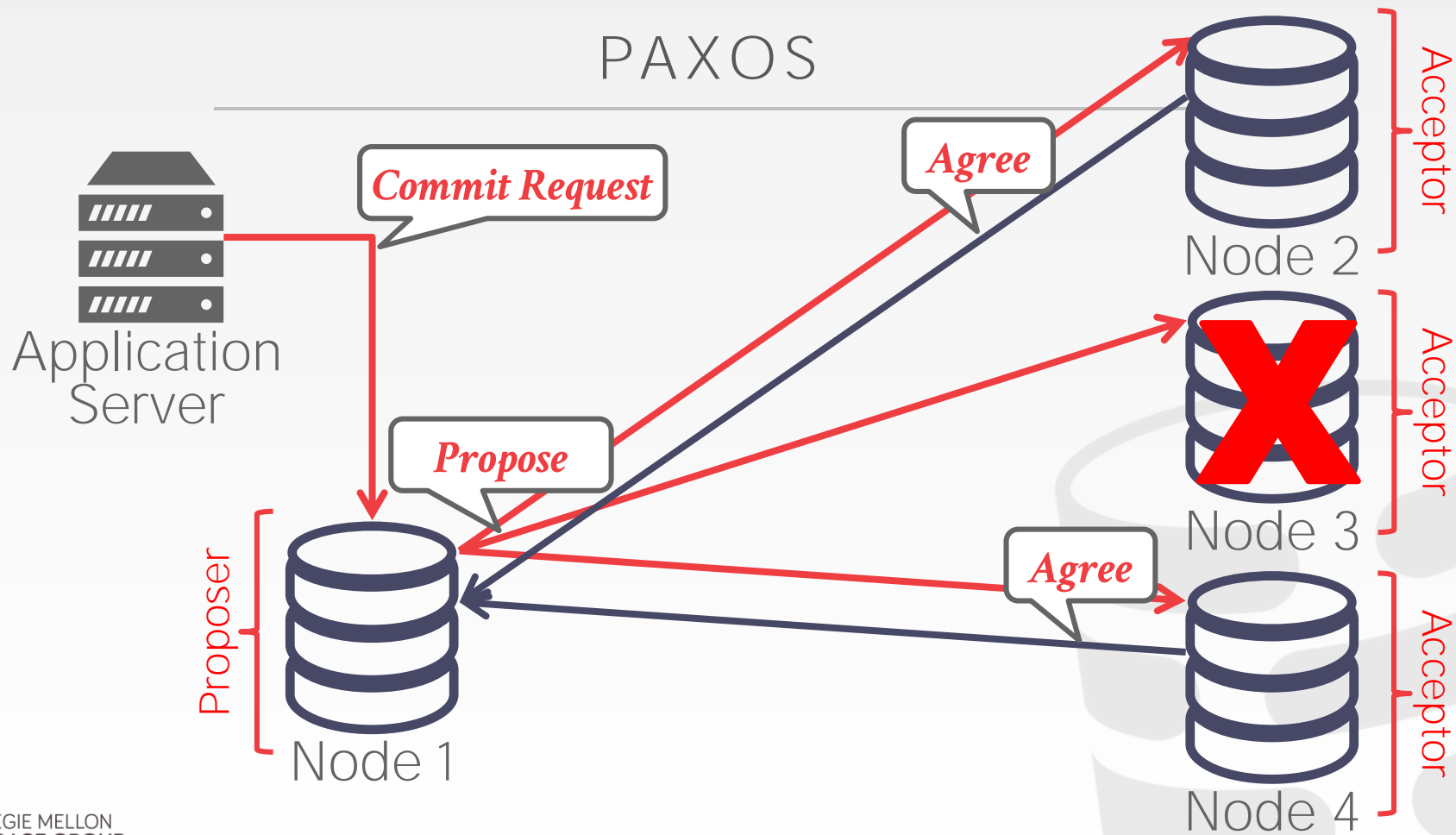


## PAXOS



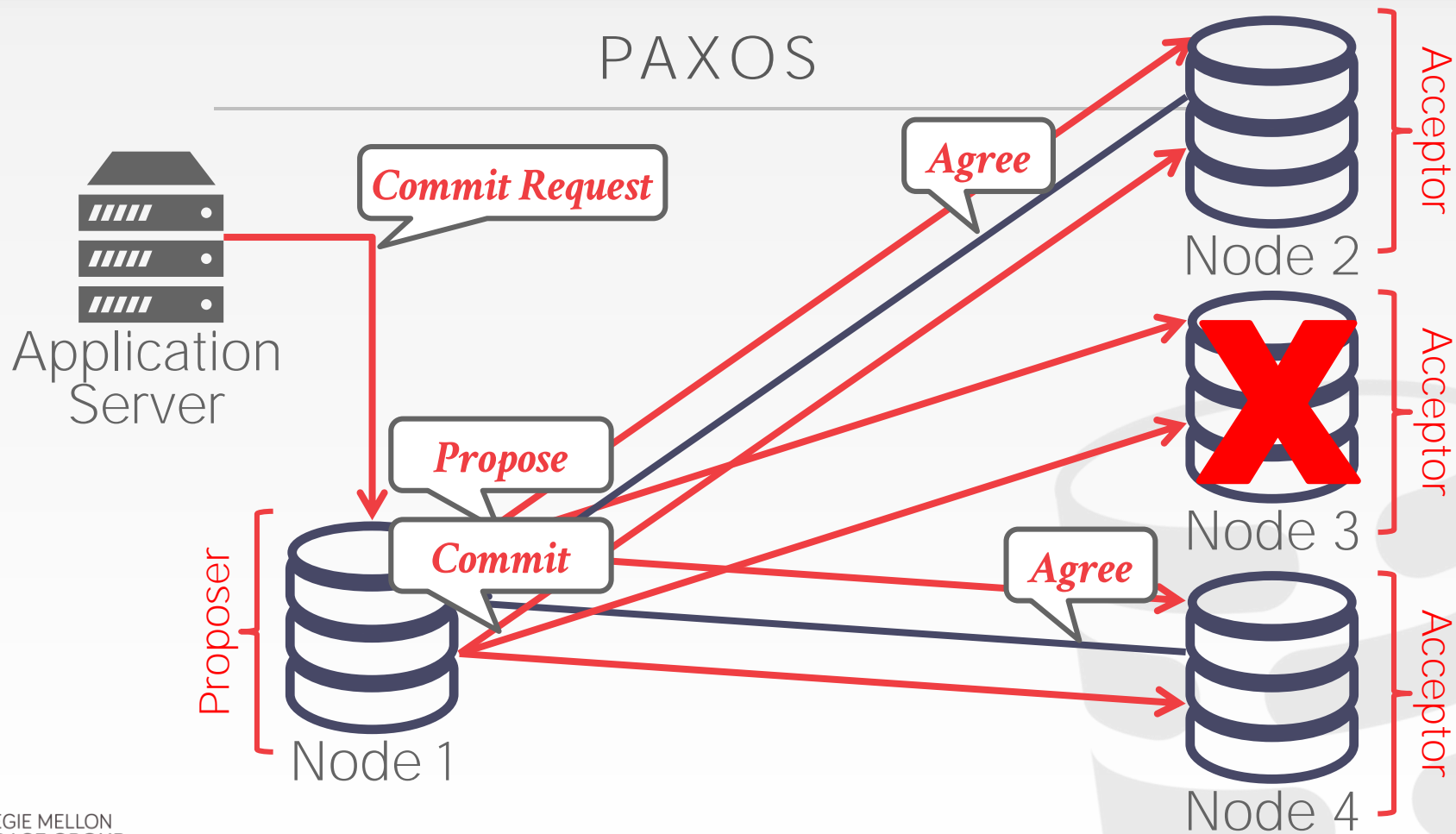
## PAXOS



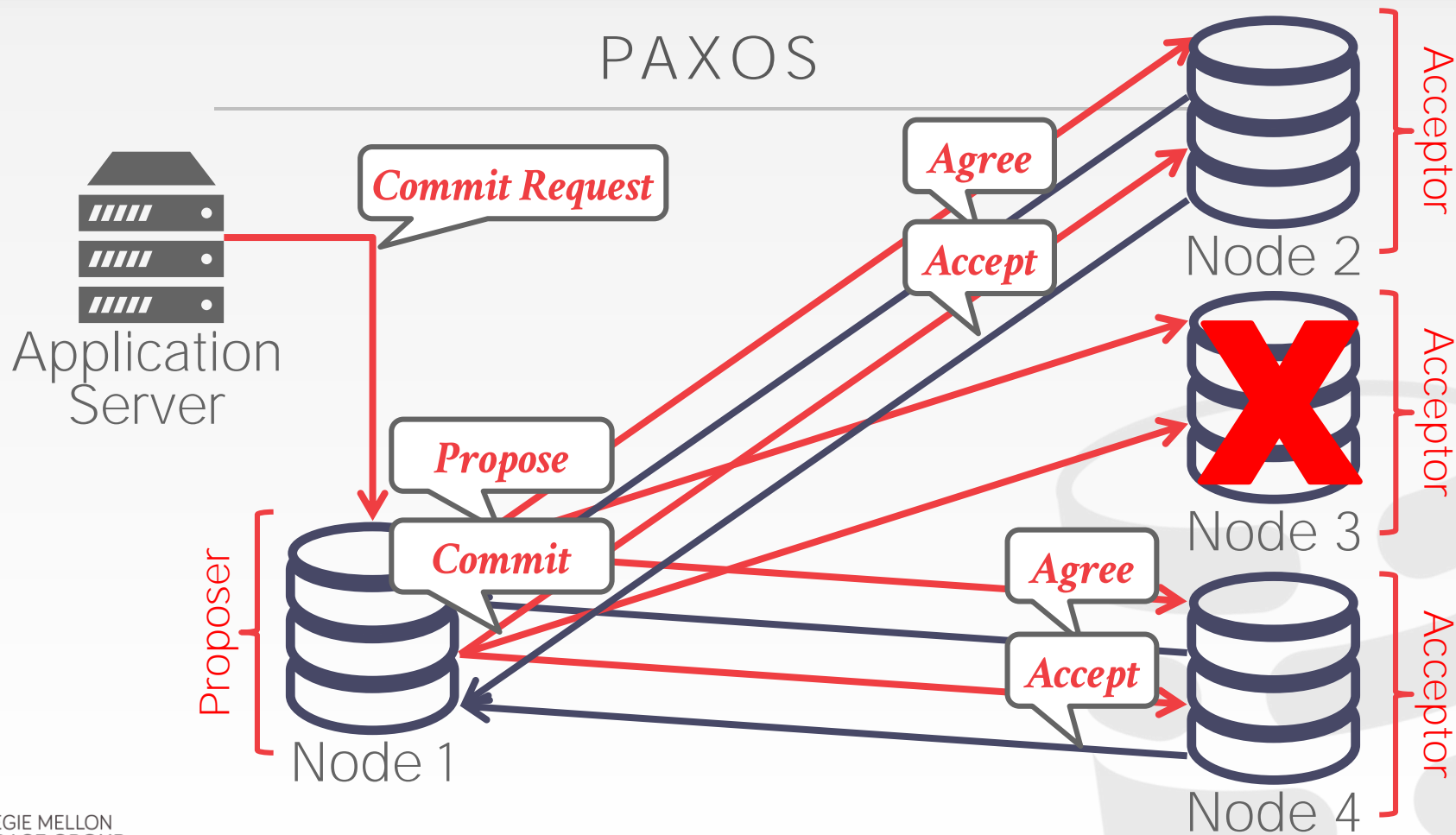




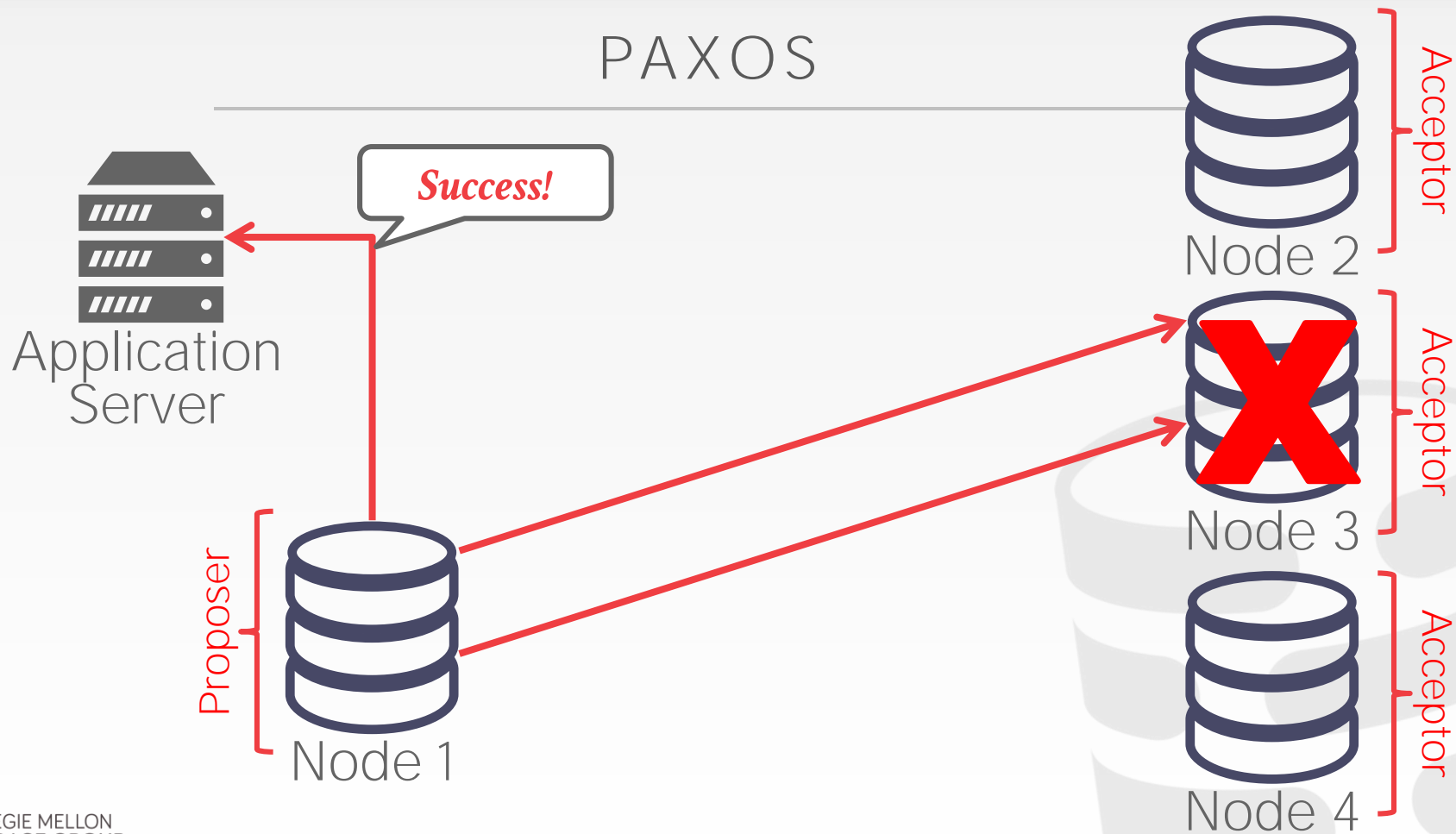
## PAXOS



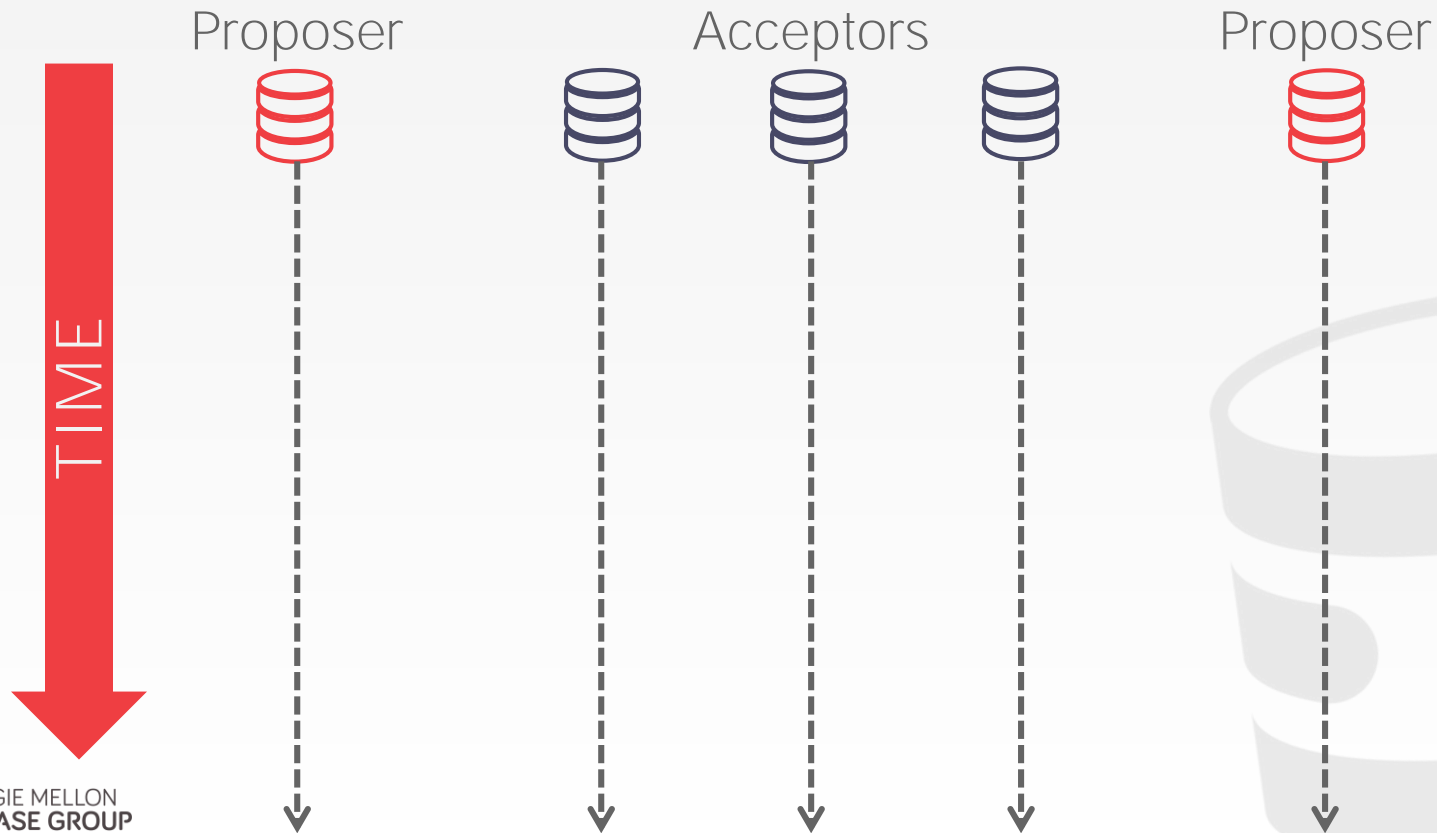
## PAXOS



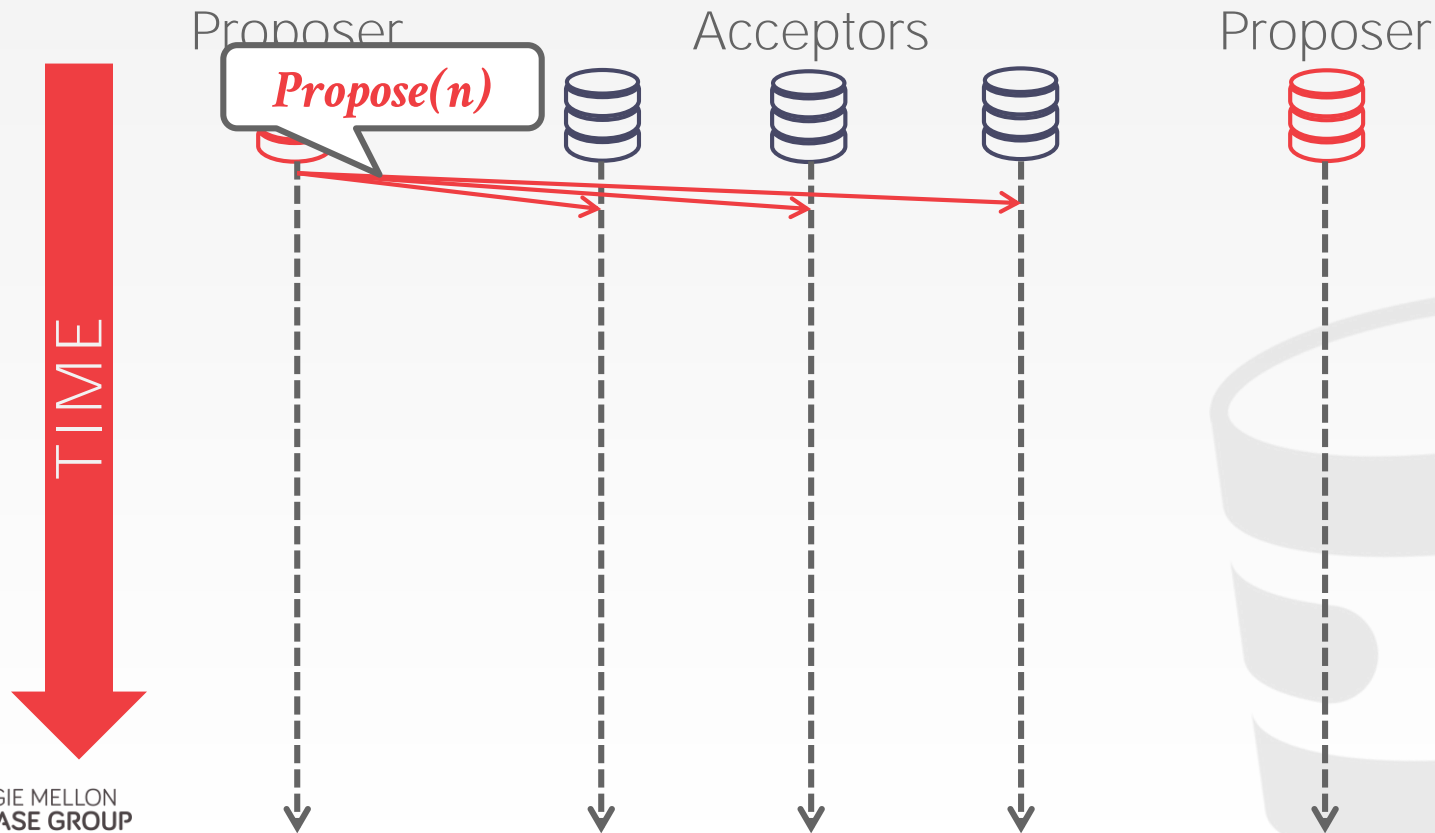
## PAXOS



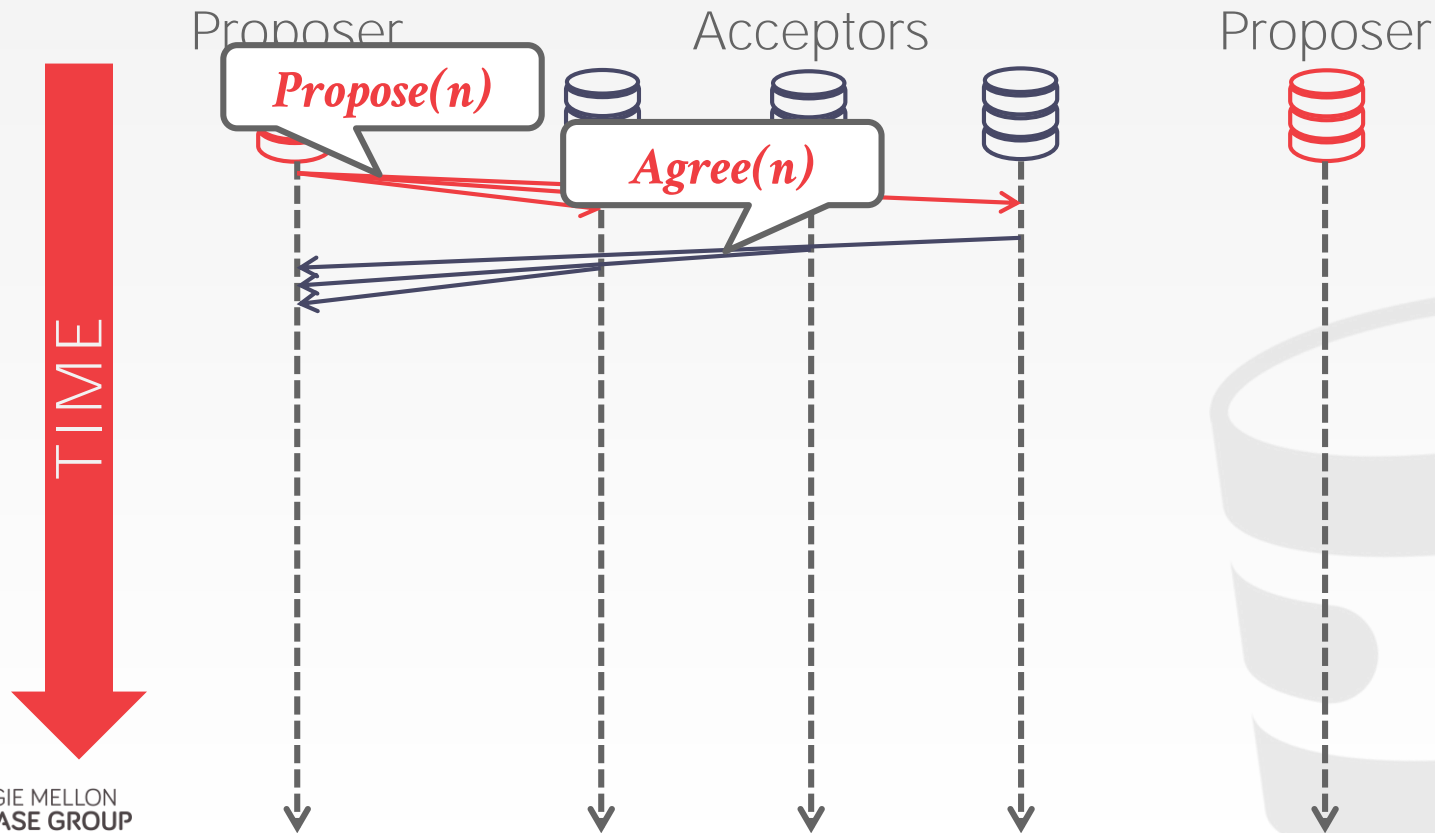
# PAXOS



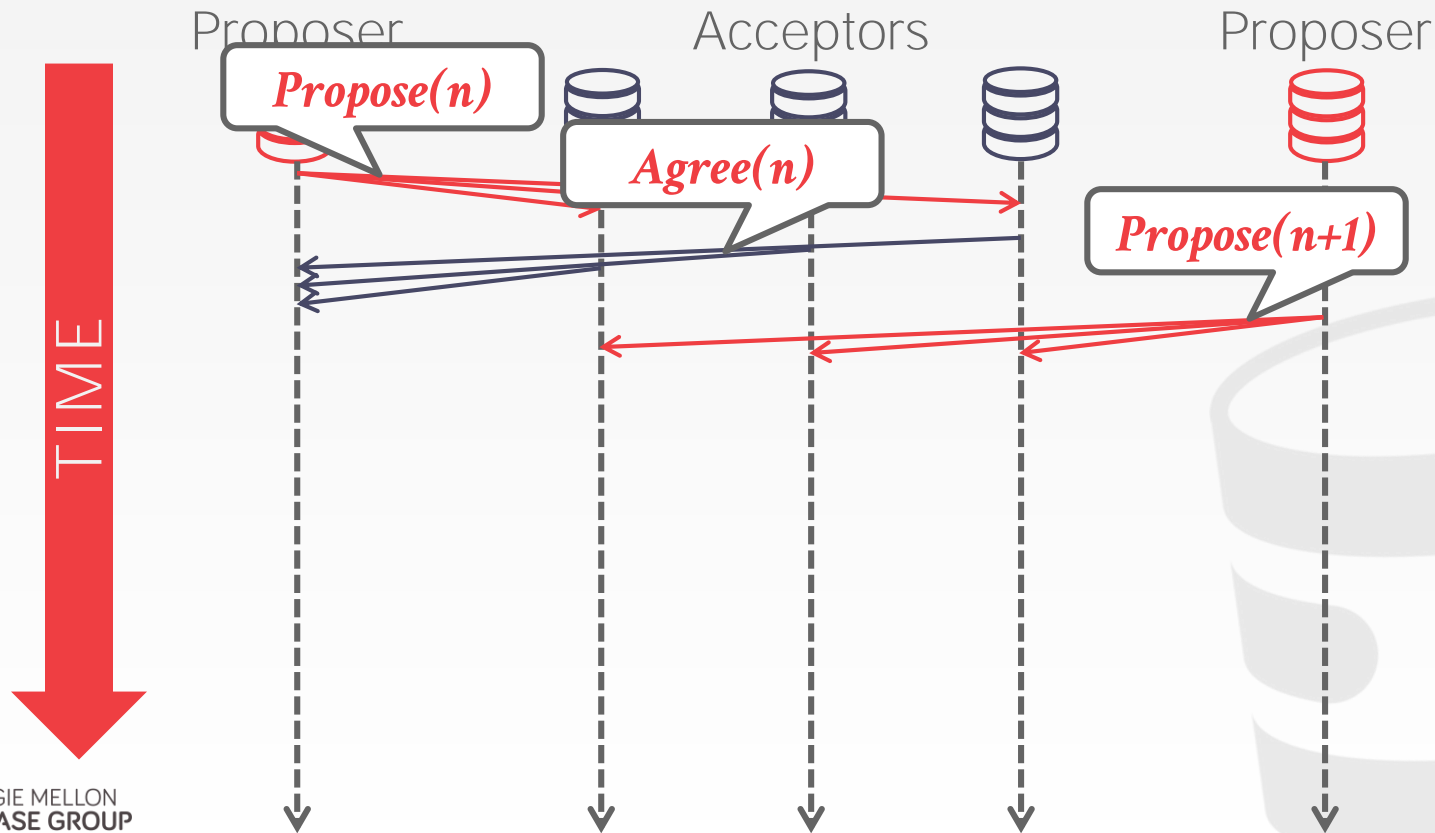
# PAXOS



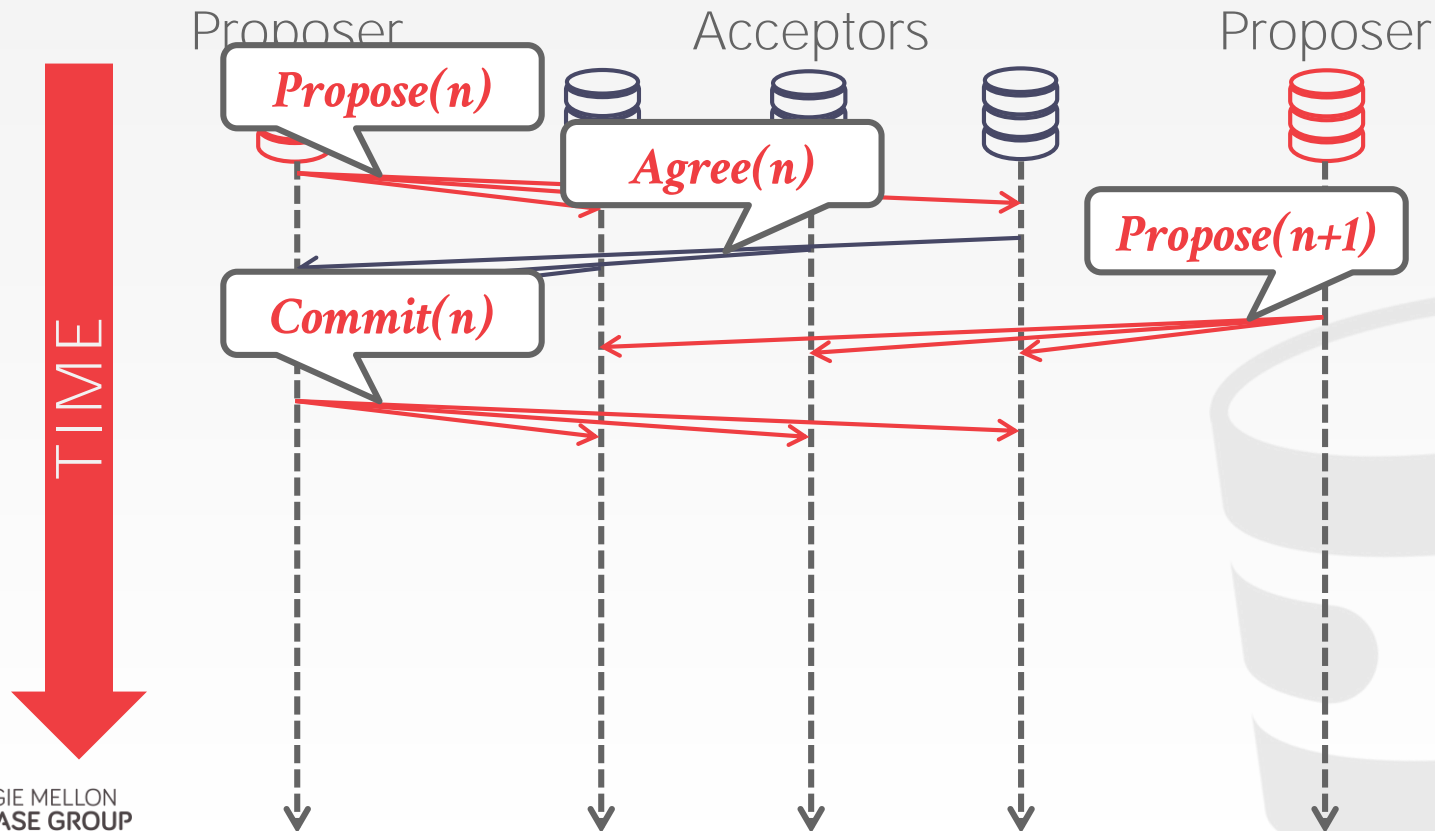
# PAXOS



# PAXOS

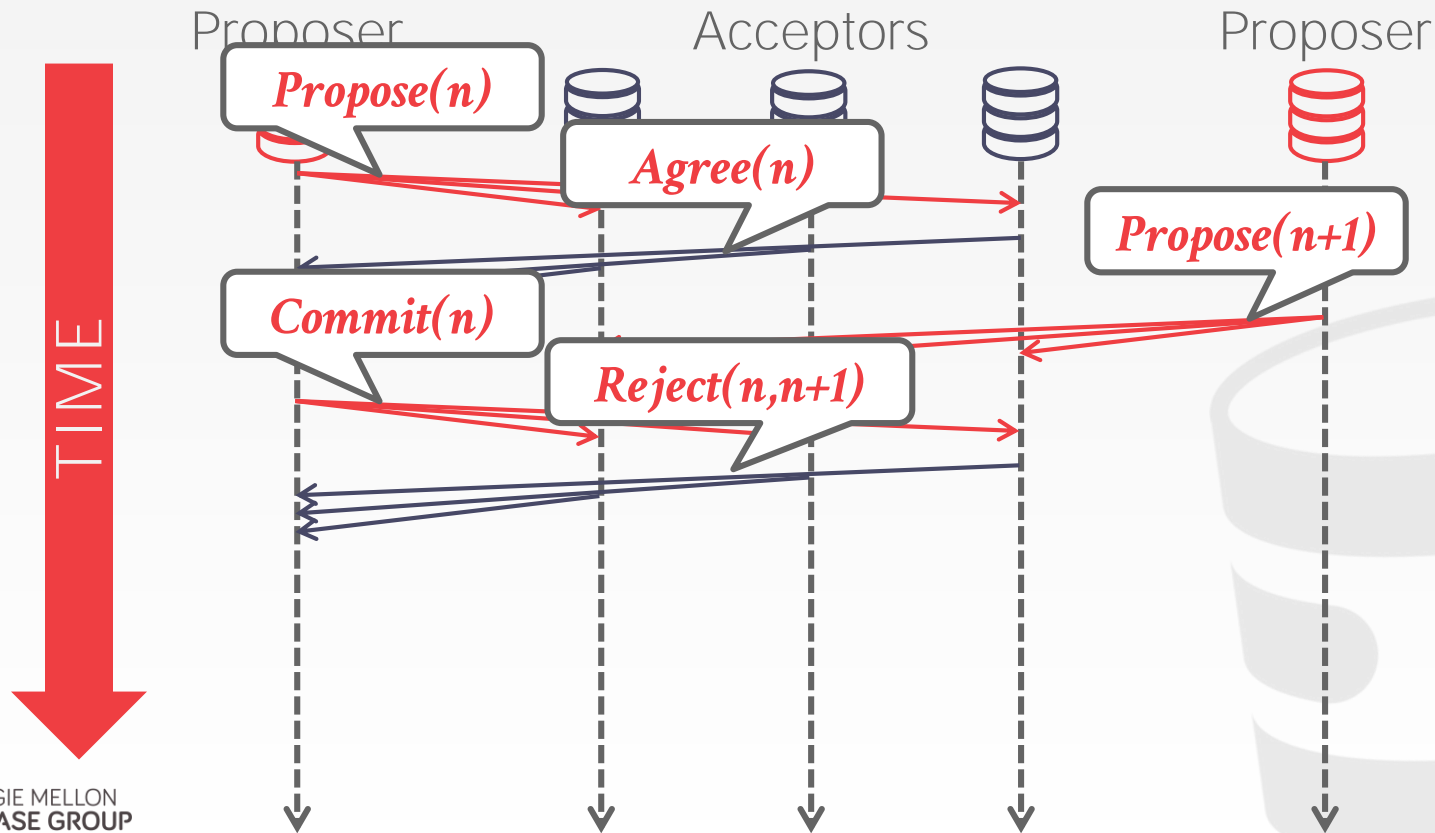


# PAXOS

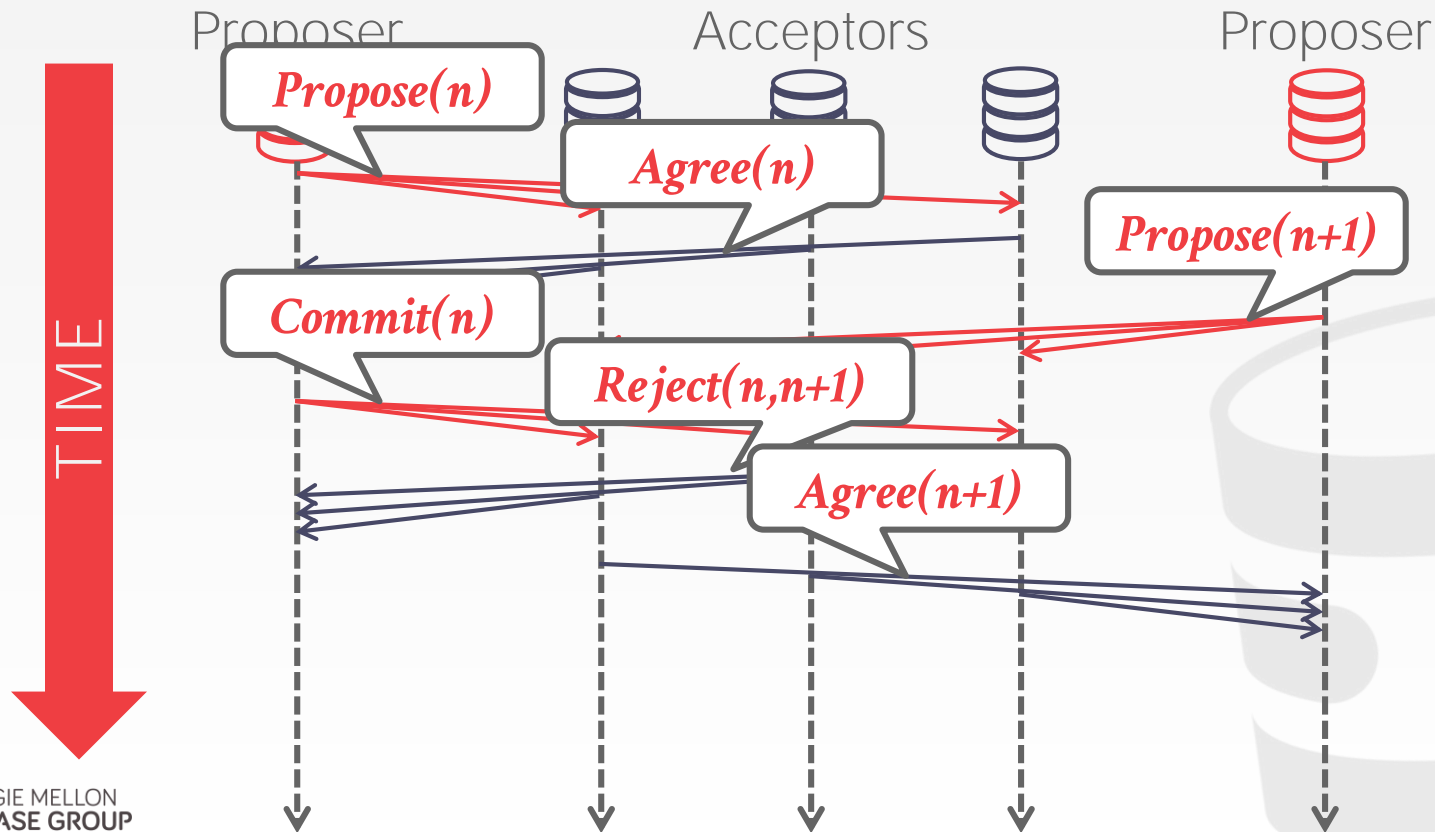




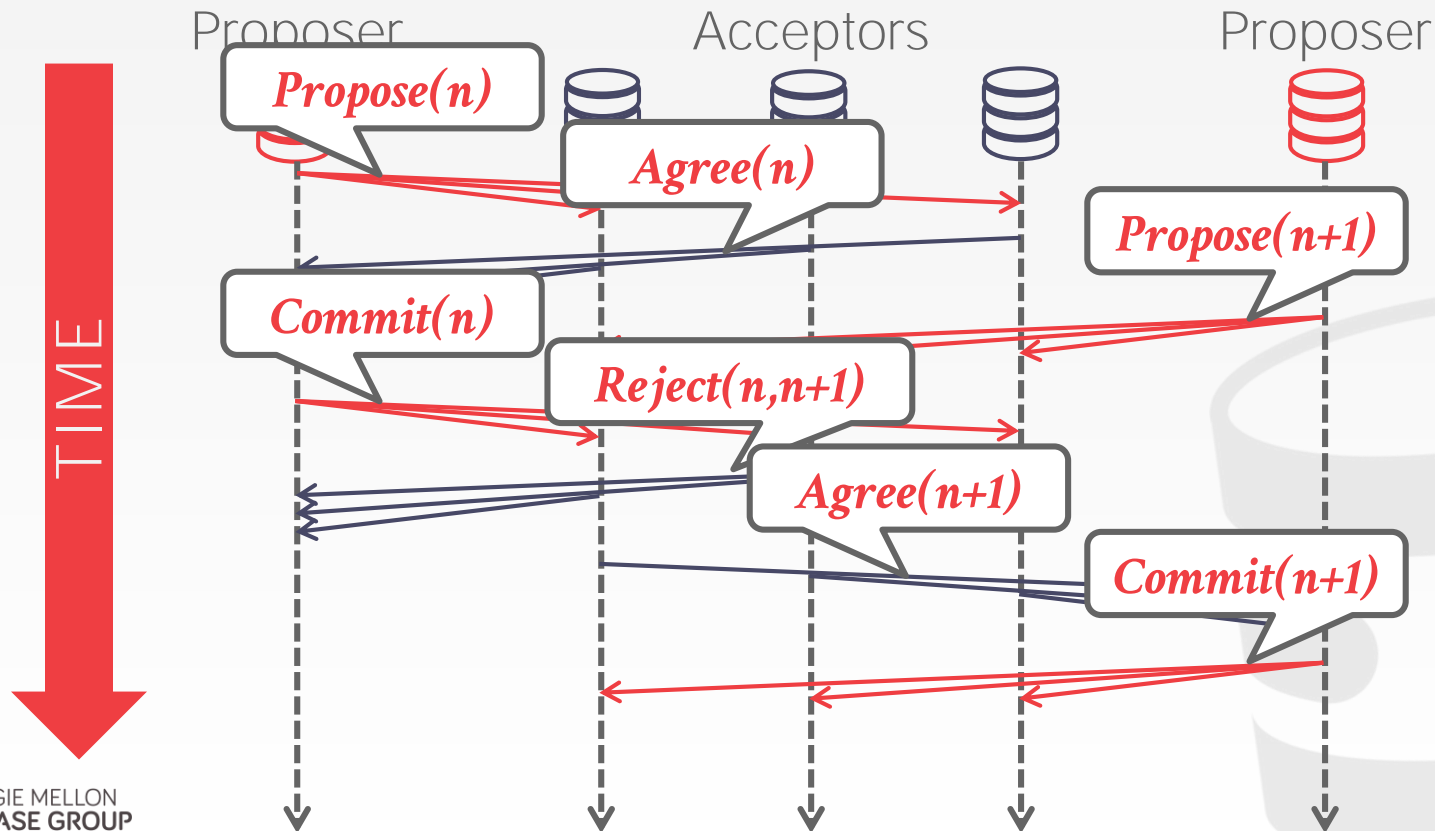
# PAXOS



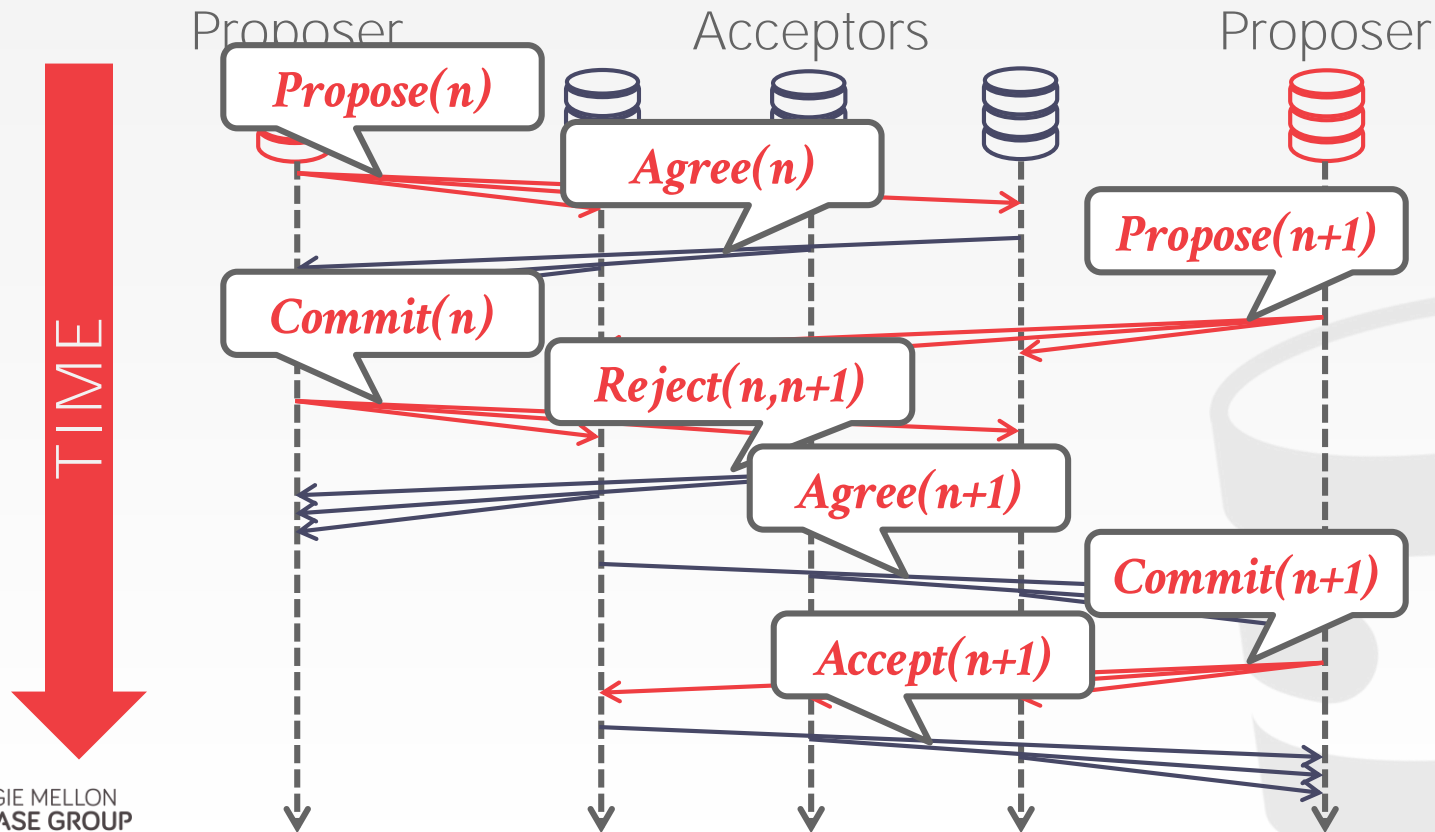
# PAXOS



# PAXOS



## PAXOS



# MULTI-PAXOS

---

If the system elects a single leader that is in charge of proposing changes for some period of time, then it can skip the **PREPARE** phase.

→ Fall back to full Paxos whenever there is a failure.

The system has to periodically renew who the leader is.

# 2PC VS. PAXOS

---

## Two-Phase Commit

- Blocks if coordinator fails after the prepare message is sent, until coordinator recovers.

## Paxos

- Non-blocking as long as a majority participants are alive, provided there is a sufficiently long period without further failures.

# REPLICATION

---

The DBMS can replicate data across redundant nodes to increase availability.

## Design Decisions:

- Replica Configuration
- Propagation Scheme
- Propagation Timing



# REPLICA CONFIGURATIONS

---

## **Approach #1: Master-Replica**

- All updates go to a designated master for each object.
- The master then propagates those updates to its replicas.
- Read-only txns may be allowed to access replicas.
- If the master goes down, then hold an election to select a new master.

## **Approach #2: Multi-Master**

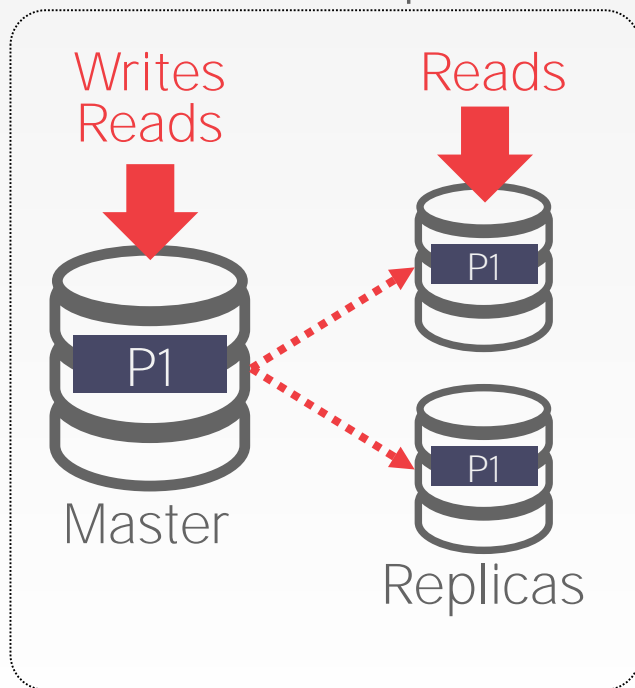
- Txns can update data objects at any replica.
- Replicas synchronize with each other.



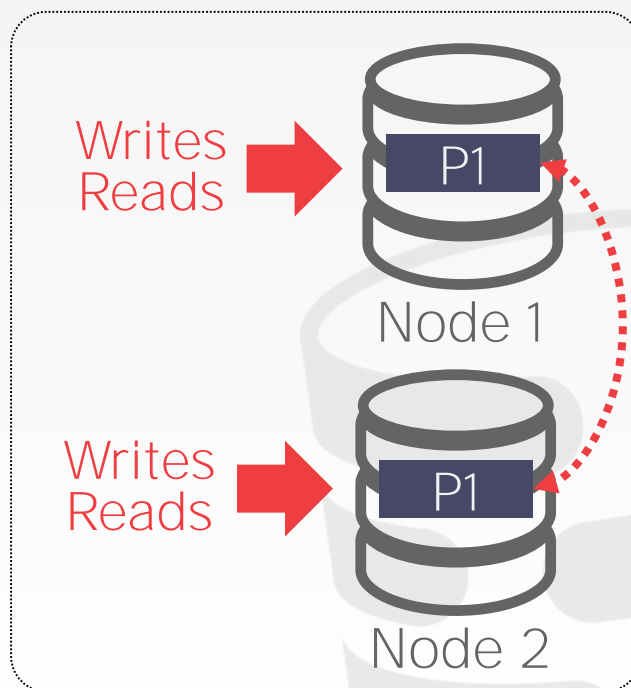


# REPLICA CONFIGURATIONS

## Master-Replica



## Multi-Master



# K-SAFETY

---

*K*-safety is a threshold for determining the fault tolerance of the replicated database.

The value *K* represents the number of replicas per data object that must exist at all times.

If the number of replicas goes below this threshold, then the DBMS halts execution and takes itself offline.

# PROPAGATION SCHEME

---

When a txn commits on a replicated database, the DBMS has to decide whether it has to wait for that txn's changes to propagate to other nodes before it can send the acknowledgement to application.

Propagation levels:

- Synchronous
- Asynchronous
- Semi-Synchronous



# PROPAGATION SCHEME

## Approach #1: Synchronous

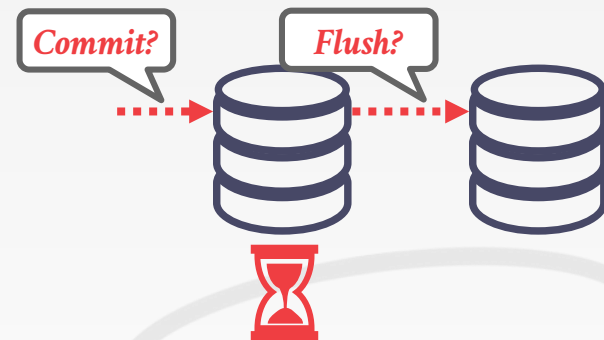
→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



# PROPAGATION SCHEME

## Approach #1: Synchronous

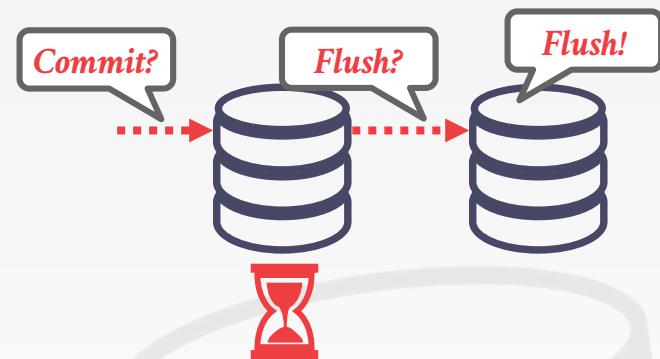
→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



# PROPAGATION SCHEME

## Approach #1: Synchronous

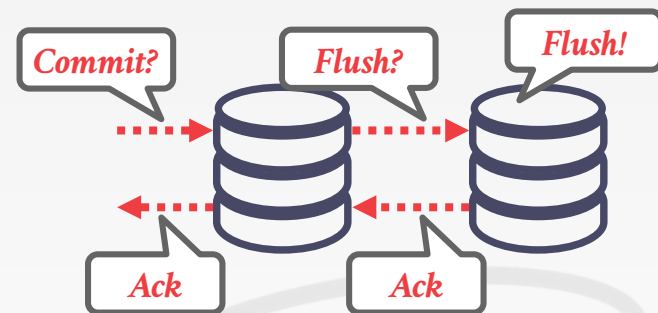
→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



# PROPAGATION SCHEME

## Approach #1: Synchronous

→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



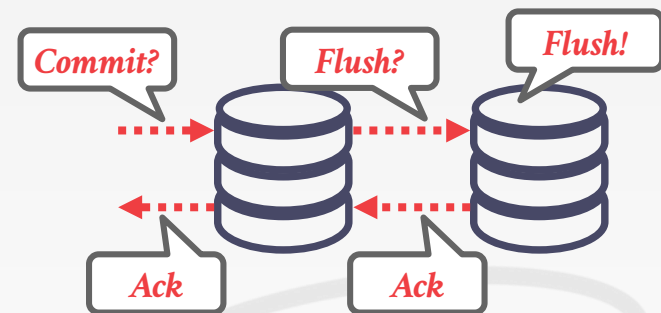
# PROPAGATION SCHEME

## Approach #1: Synchronous

→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.

## Approach #2: Asynchronous

→ The master immediately returns the acknowledgement to the client without waiting for replicas to apply the changes.





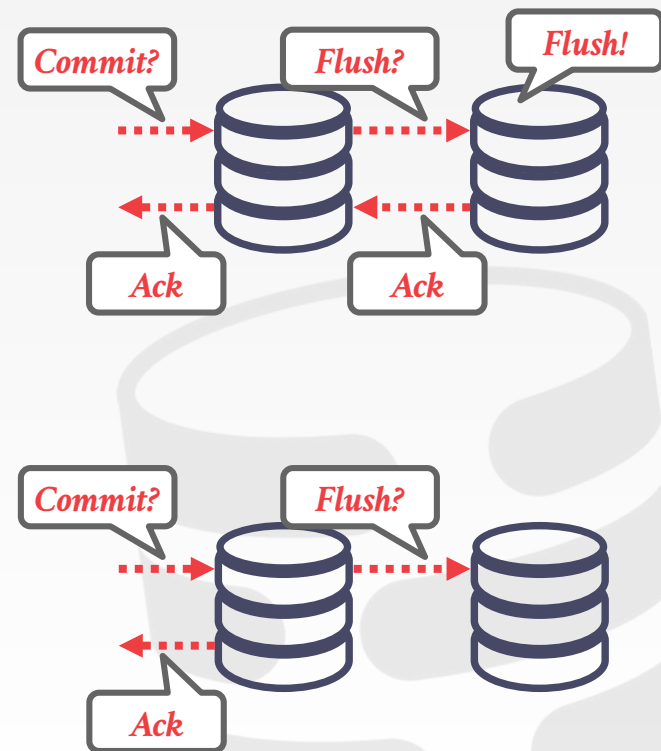
# PROPAGATION SCHEME

## Approach #1: Synchronous

→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.

## Approach #2: Asynchronous

→ The master immediately returns the acknowledgement to the client without waiting for replicas to apply the changes.



# PROPAGATION SCHEME

## Approach #3: Semi-Synchronous

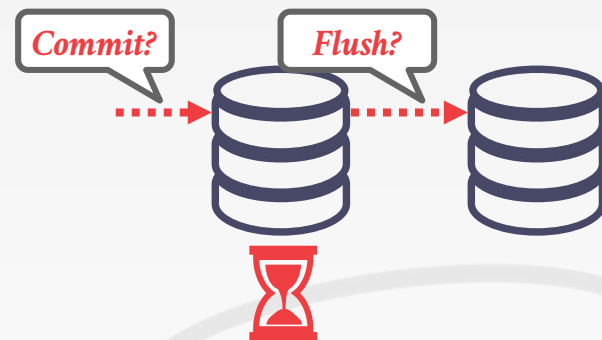
→ Replicas immediately send acknowledgements without logging them.



# PROPAGATION SCHEME

## Approach #3: Semi-Synchronous

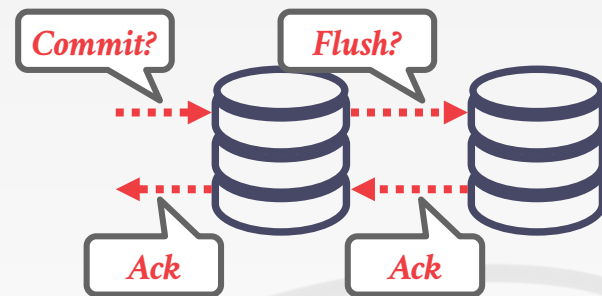
→ Replicas immediately send acknowledgements without logging them.



# PROPAGATION SCHEME

## Approach #3: Semi-Synchronous

→ Replicas immediately send acknowledgements without logging them.

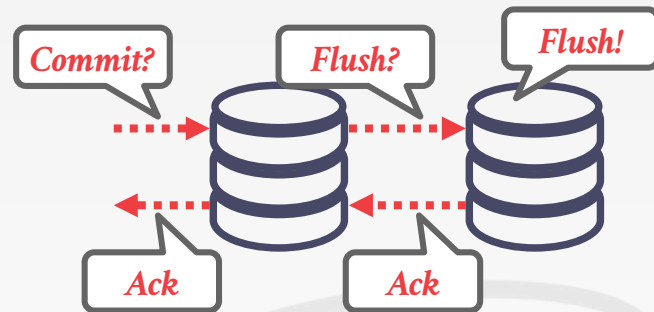


# PROPAGATION SCHEME

## Approach #3: Semi-Synchronous

→ Replicas immediately send acknowledgements without logging them.

Applications can make trade-offs on protecting the integrity of the database versus performance.



# PROPAGATION TIMING

---

## **Approach #1: Continuous**

- The DBMS sends log messages immediately as it generates them.
- Also need to send a commit/abort message.

## **Approach #2: On Commit**

- The DBMS only sends the log messages for a txn to the replicas once the txn is commits.
- Do not waste time sending log records for aborted txns.
- Assumes that a txn's log fits entirely in memory.

# ACTIVE VS. PASSIVE

---

## **Approach #1: Active-Active**

- A txn executes at each replica independently.
- Need to check at the end whether the txn ends up with the same result at each replica.

## **Approach #2: Active-Passive**

- Each txn executes at a single location and propagates the changes to the replica.
- Not the same as master-replica vs. multi-master

# CAP THEOREM

Proposed by Eric Brewer that it is impossible for a distributed system to always be:

- Consistent
- Always Available
- Network Partition Tolerant

Proved in 2002.

*Pick Two!  
Sort of...*



Brewer



# CAP THEOREM

Consistency  
Availability  
Partition Tolerant

*Linearizability*

C

*All up nodes can satisfy  
all requests.*

A

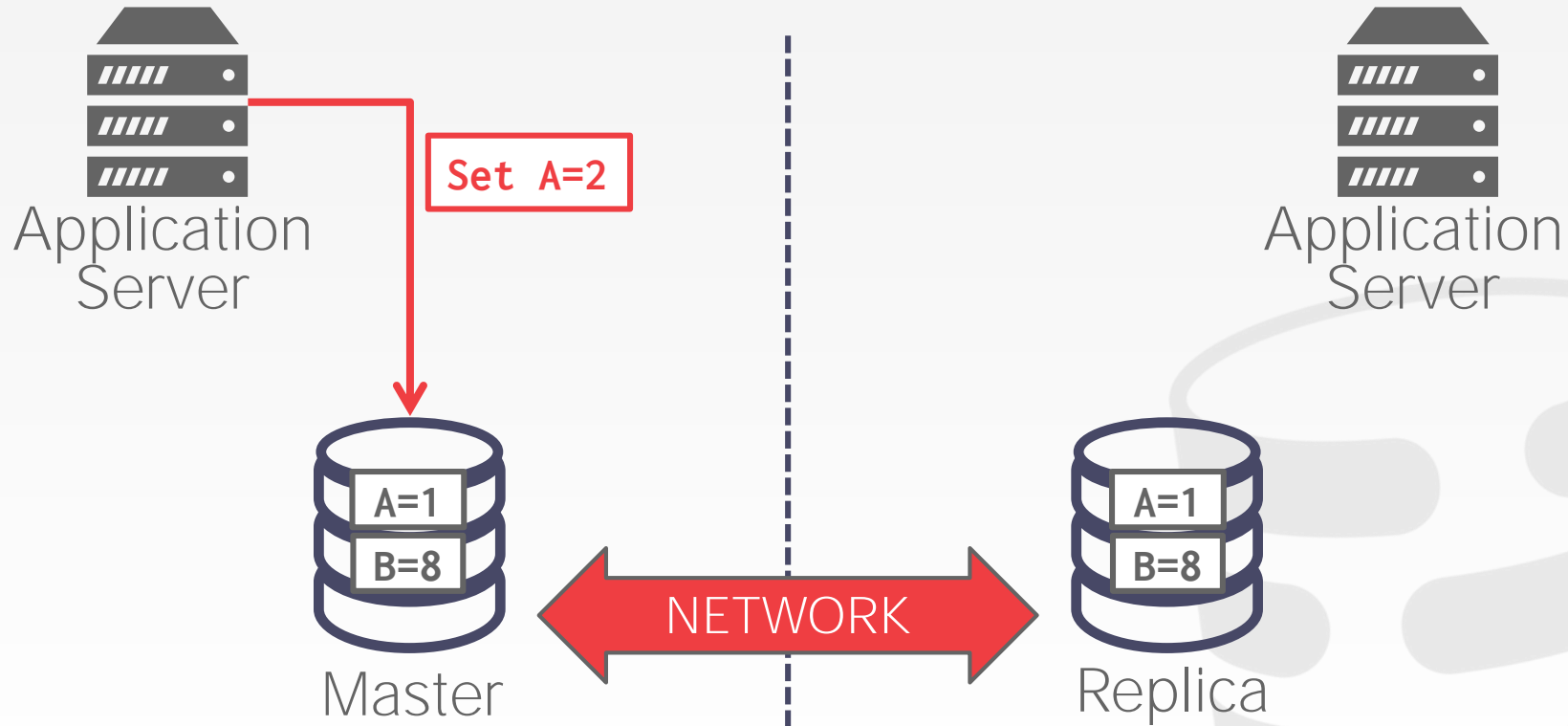


*Impossible*

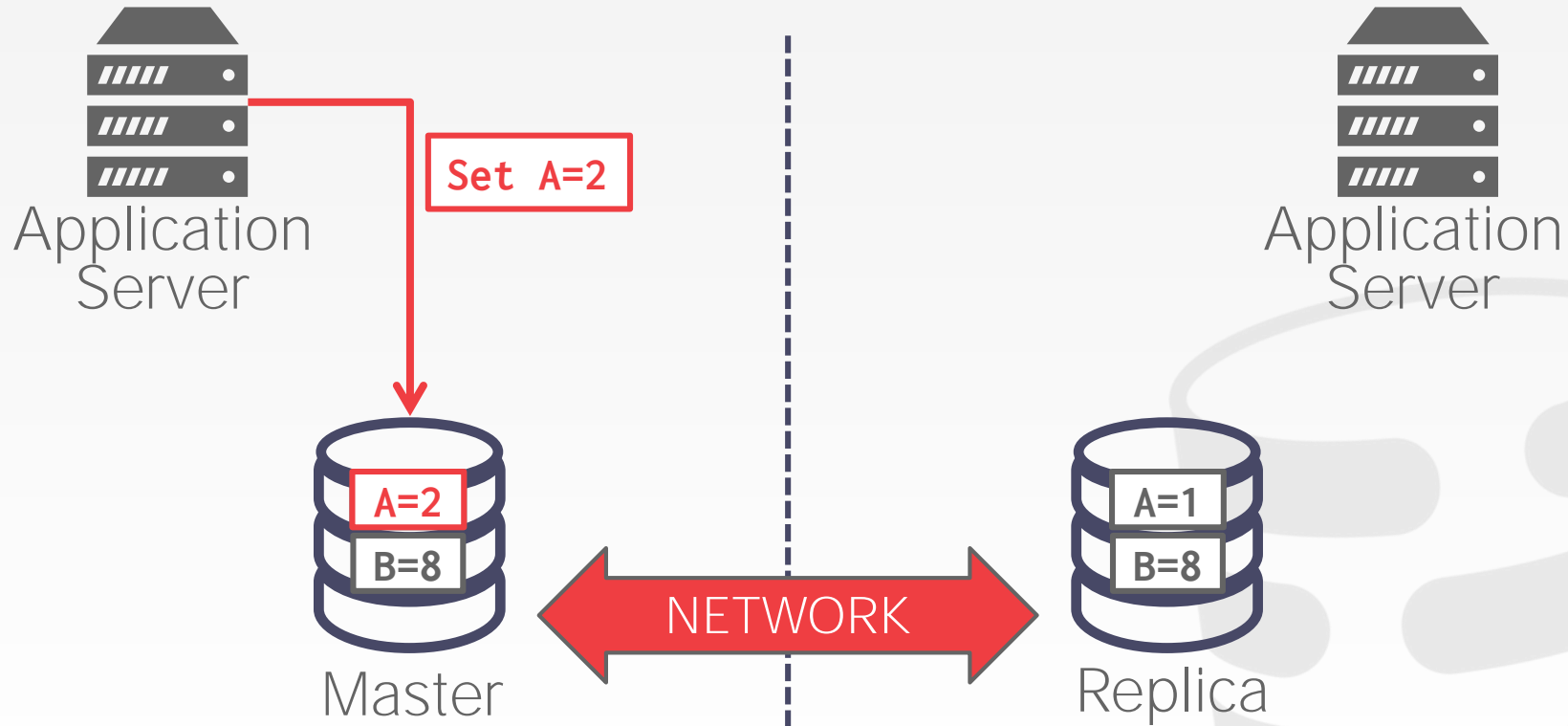
P

*Still operate correctly  
despite message loss.*

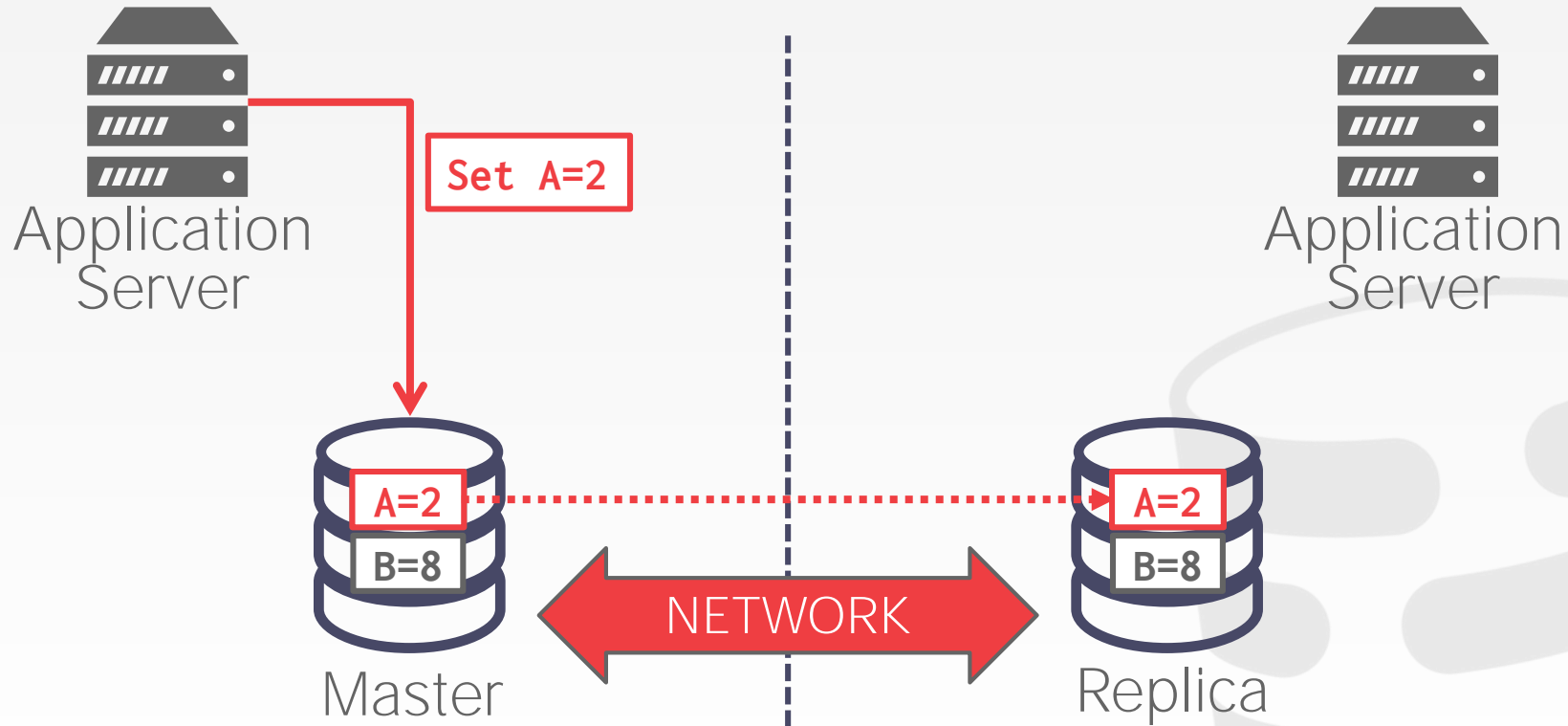
# CAP – CONSISTENCY



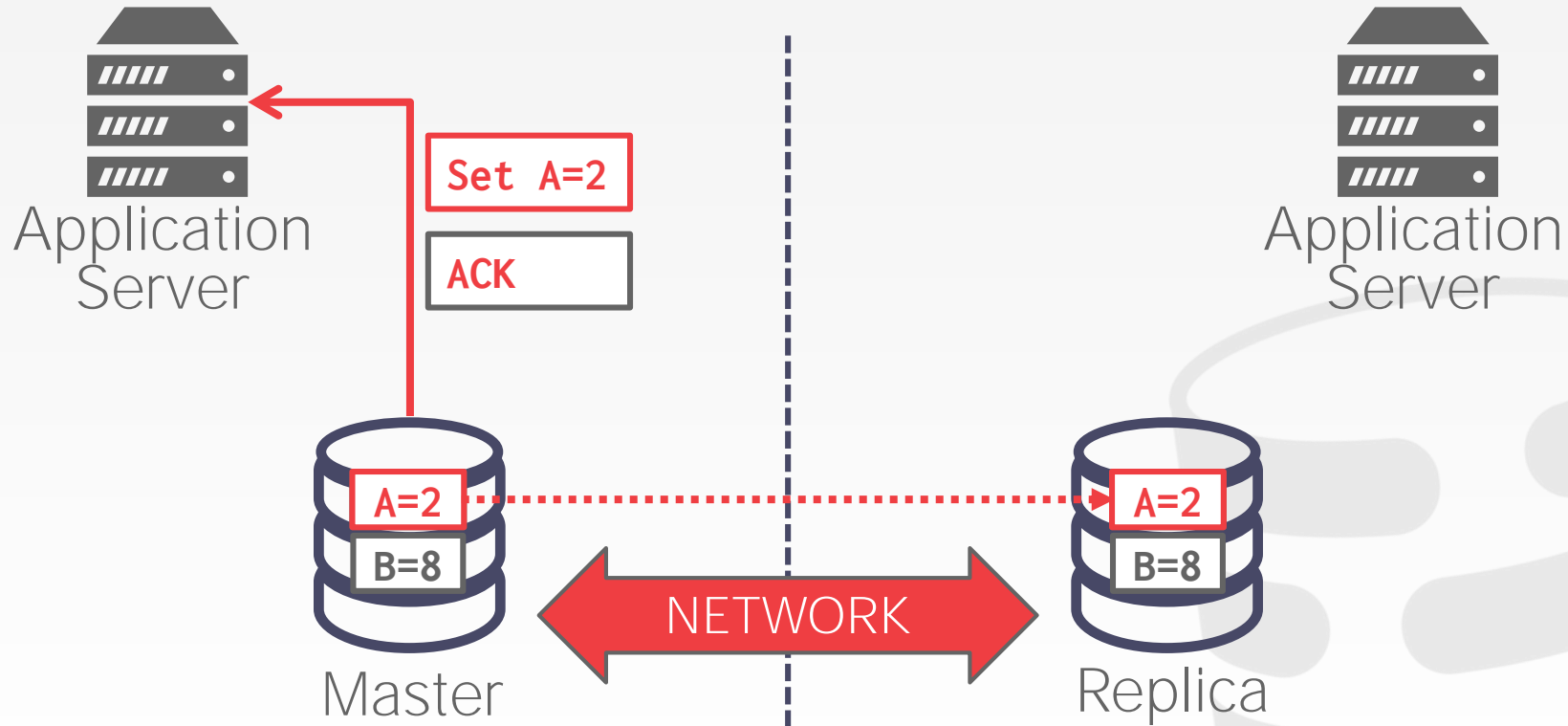
# CAP – CONSISTENCY



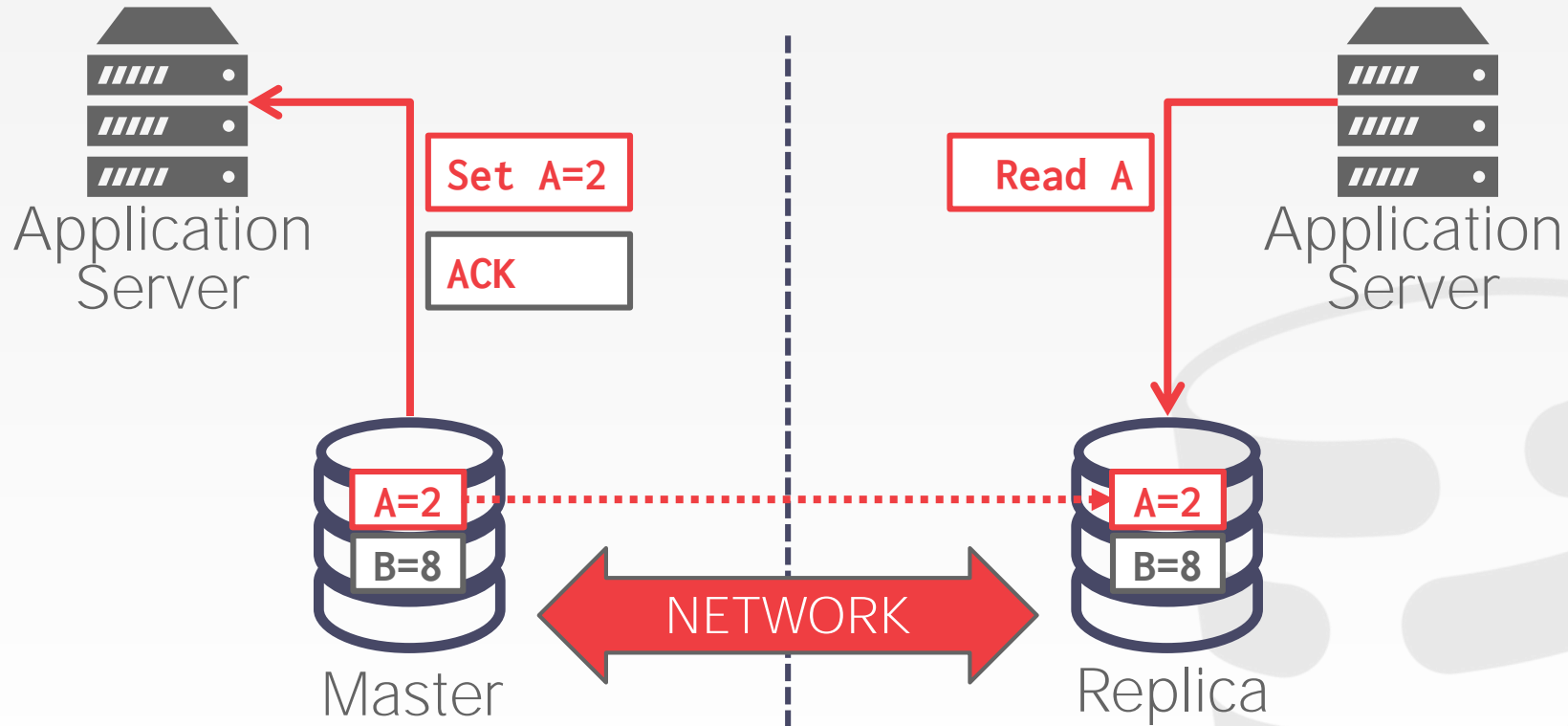
# CAP – CONSISTENCY



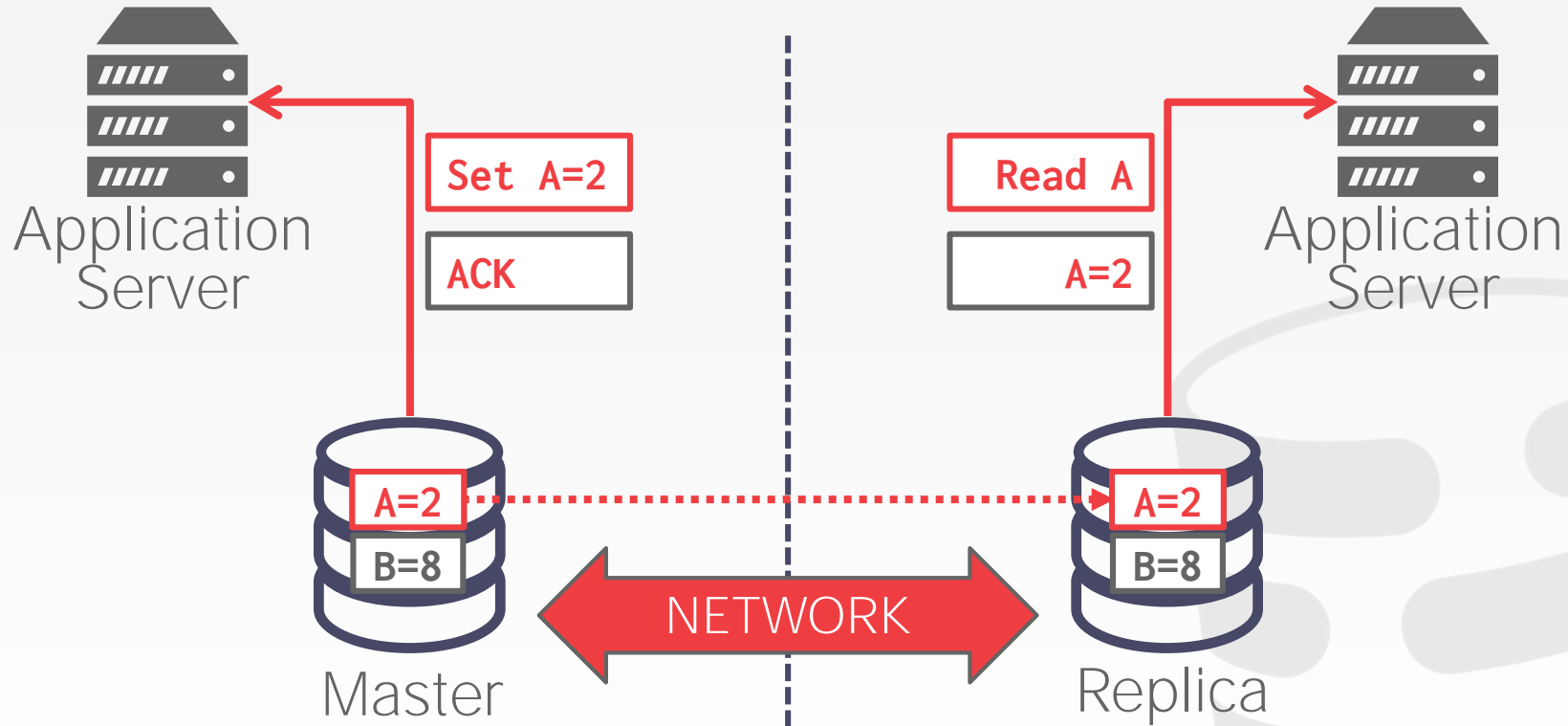
# CAP – CONSISTENCY

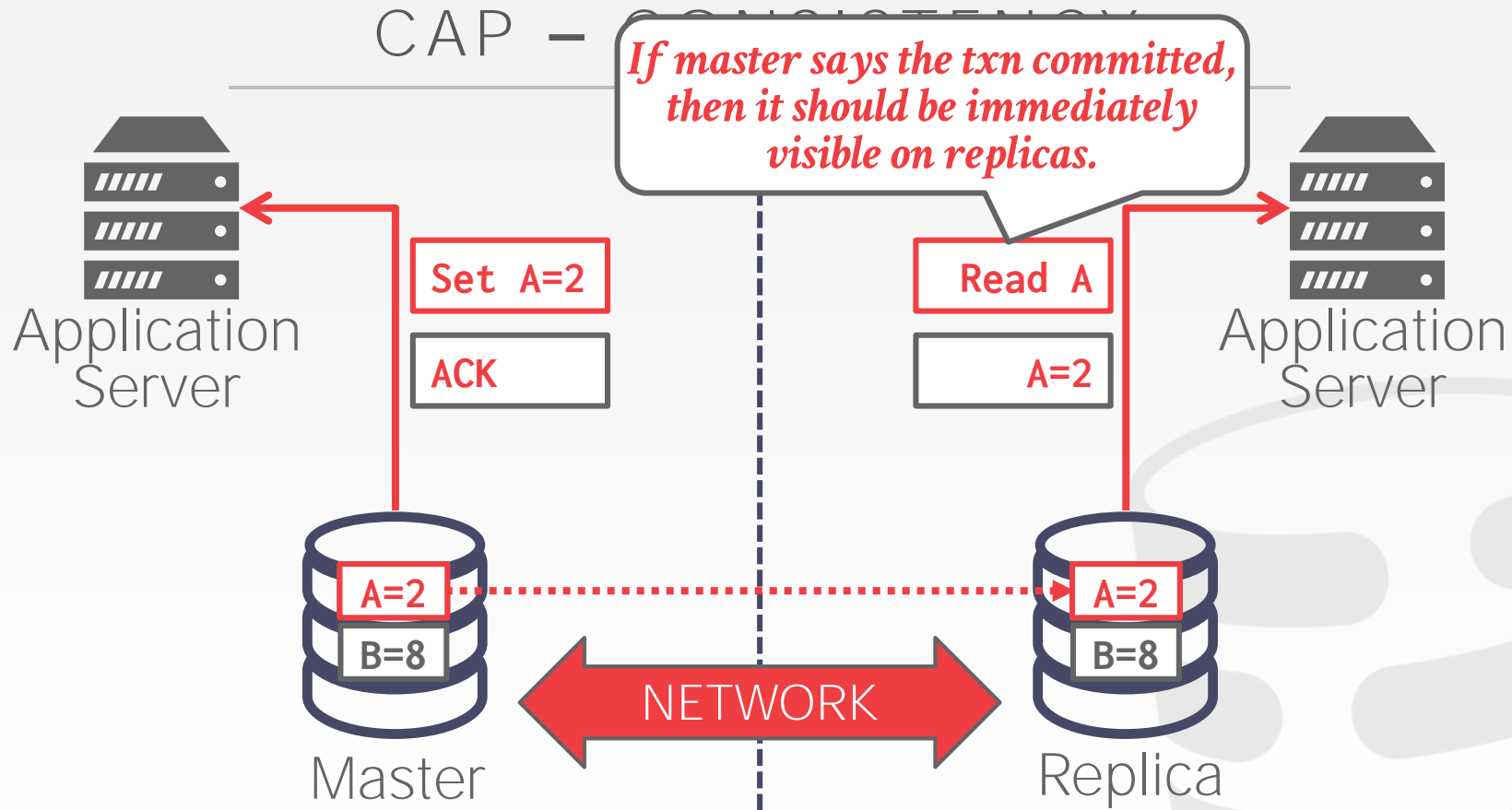


# CAP – CONSISTENCY



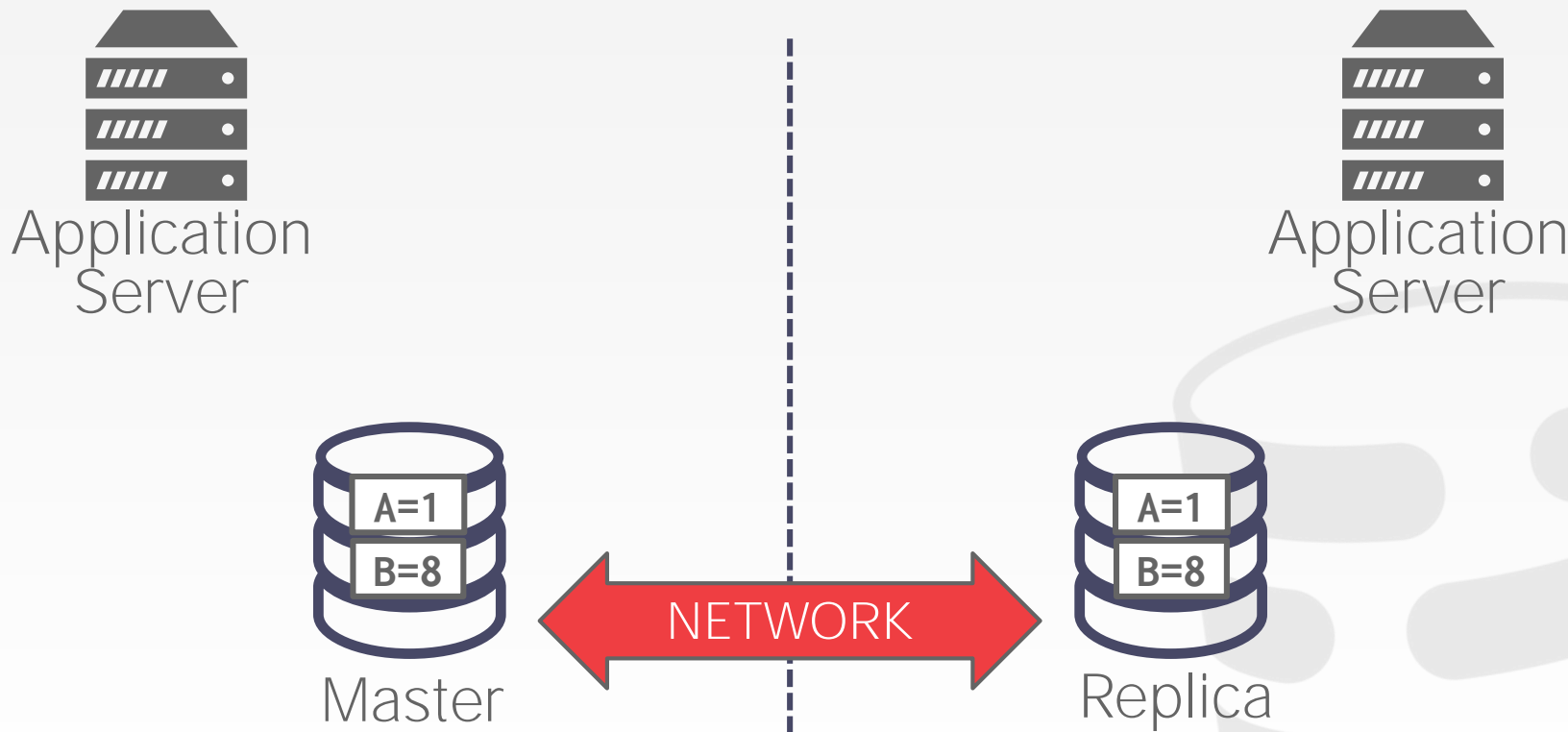
# CAP – CONSISTENCY



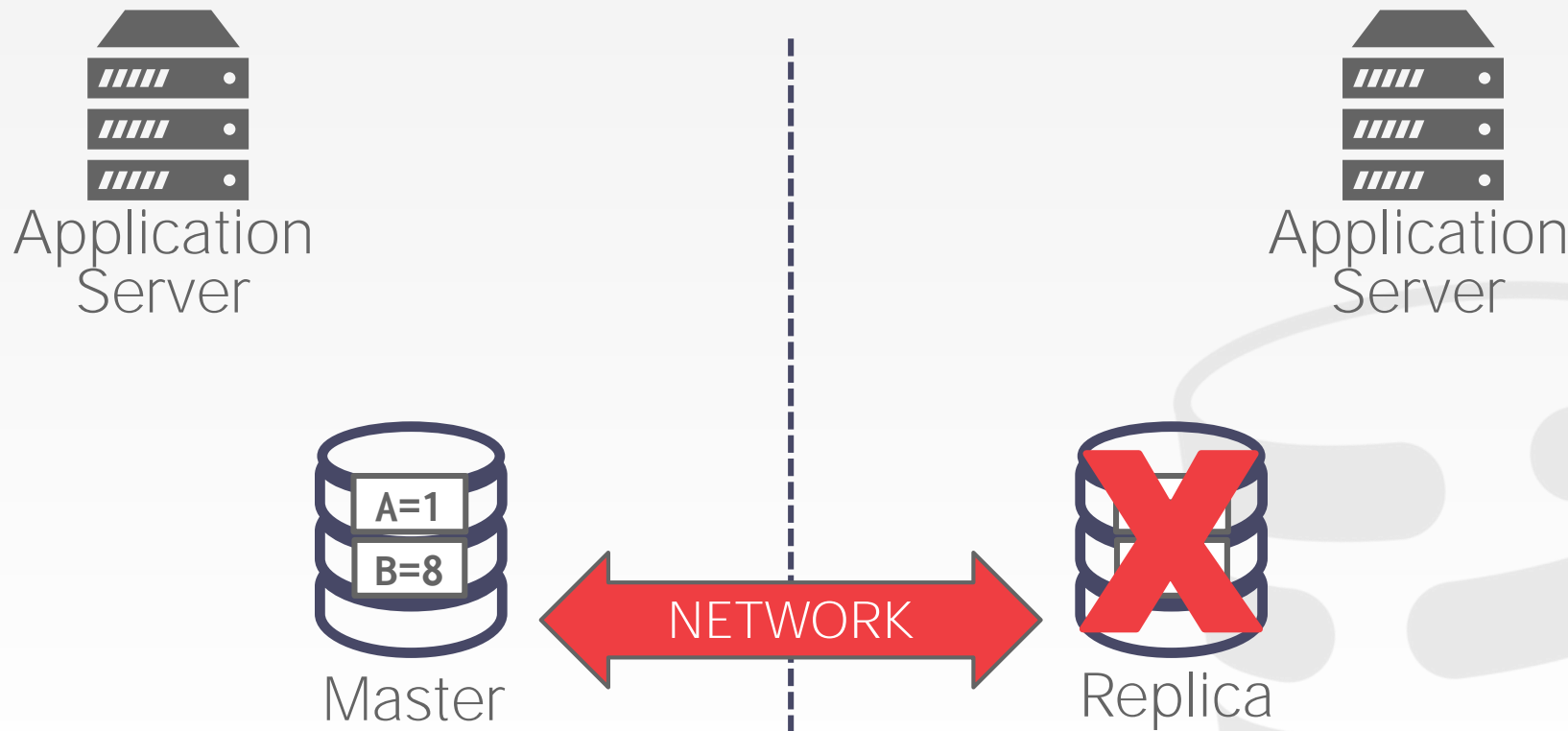




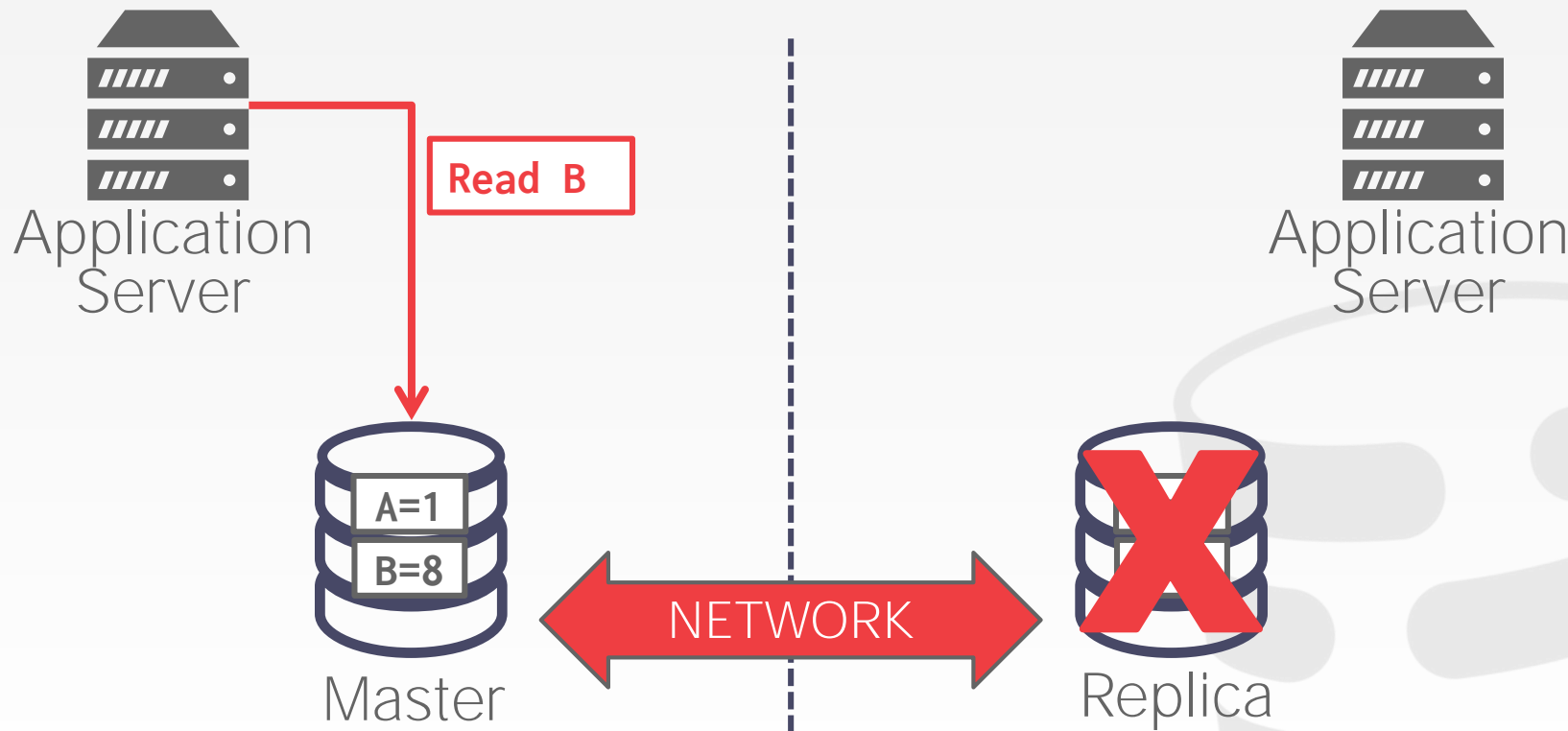
# CAP – AVAILABILITY



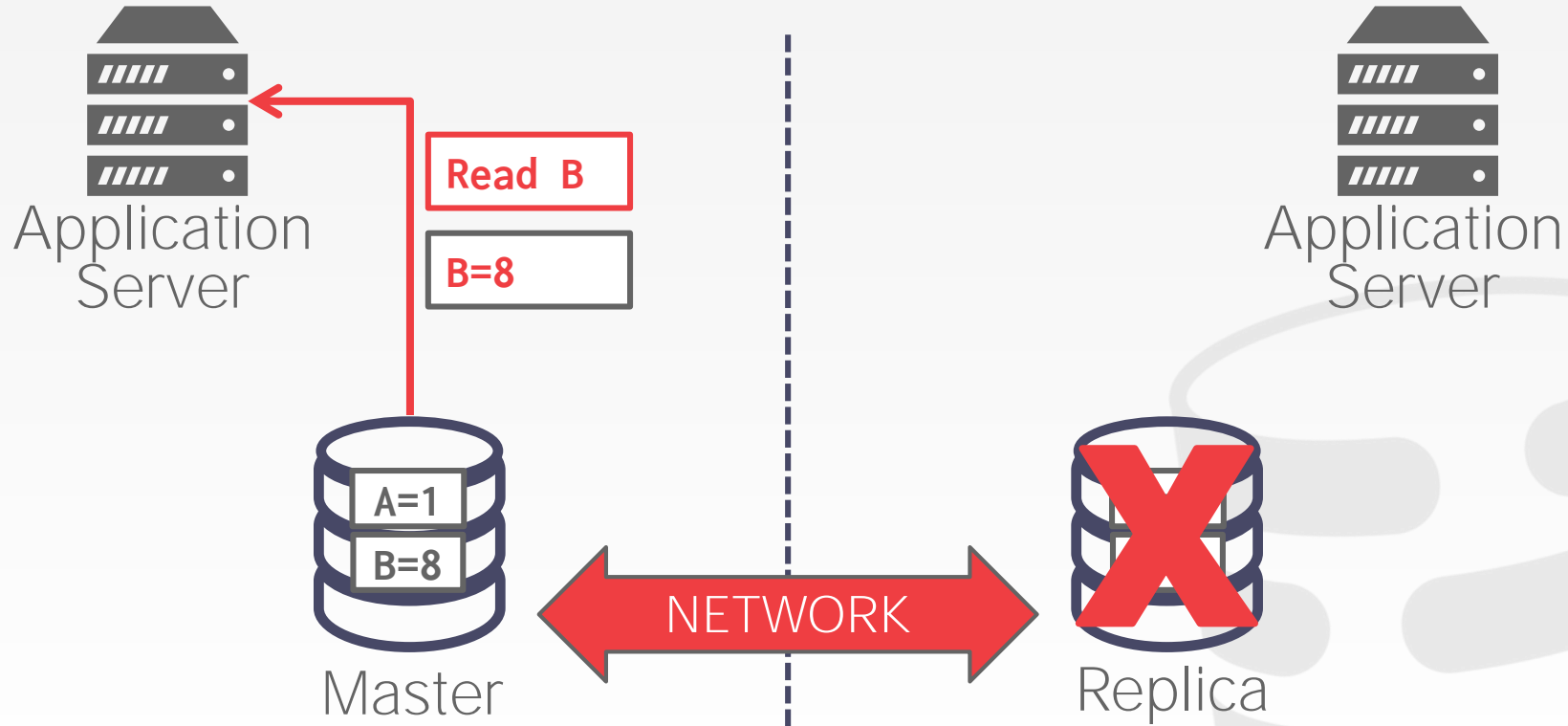
# CAP – AVAILABILITY



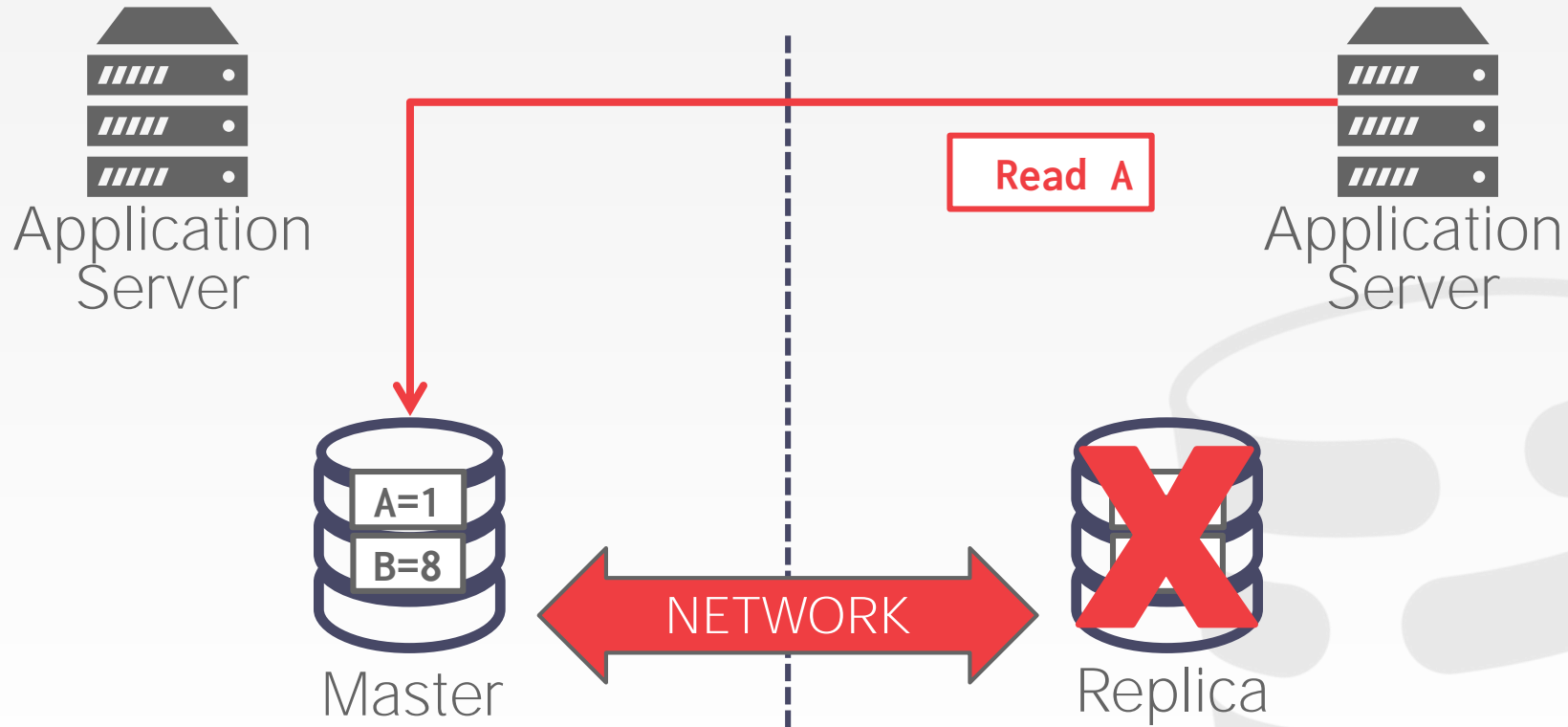
# CAP – AVAILABILITY



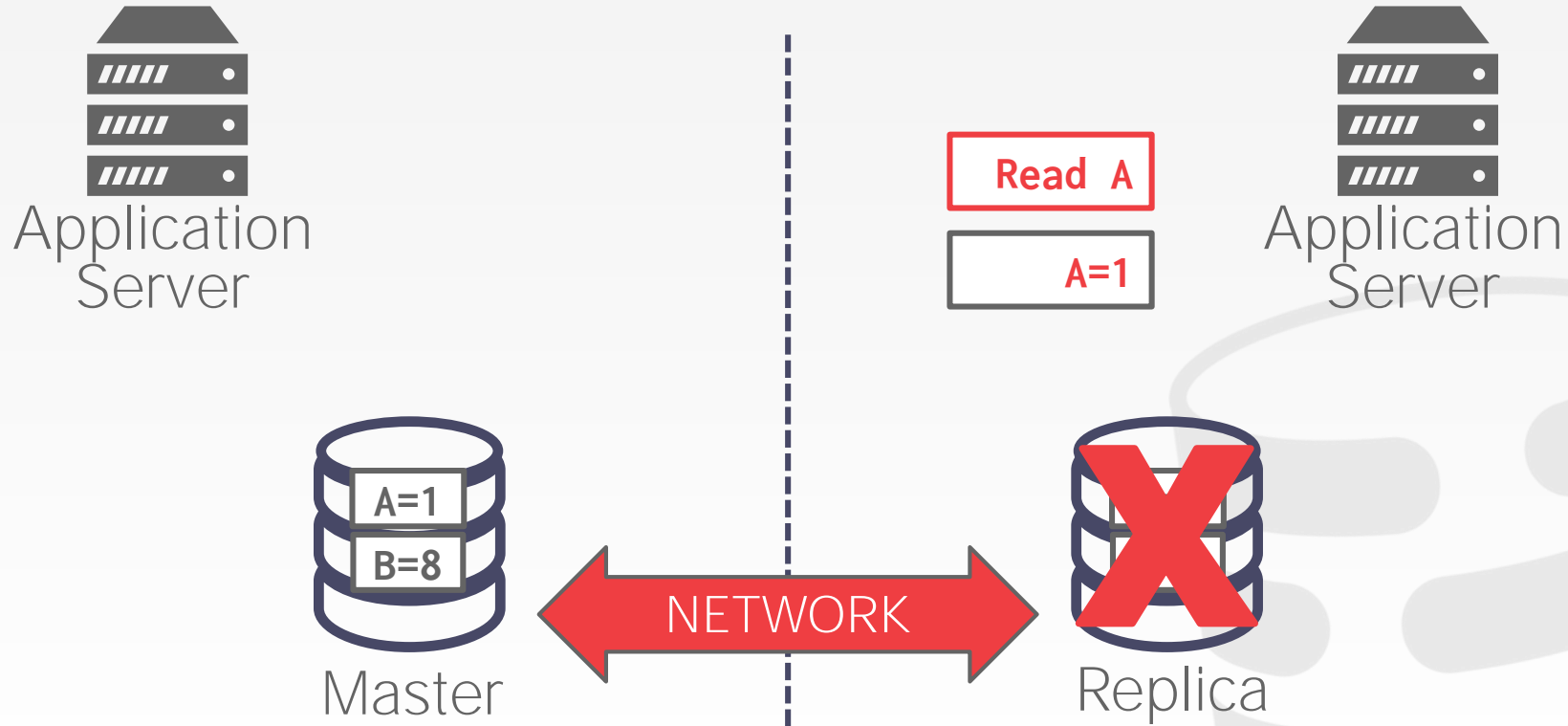
# CAP – AVAILABILITY



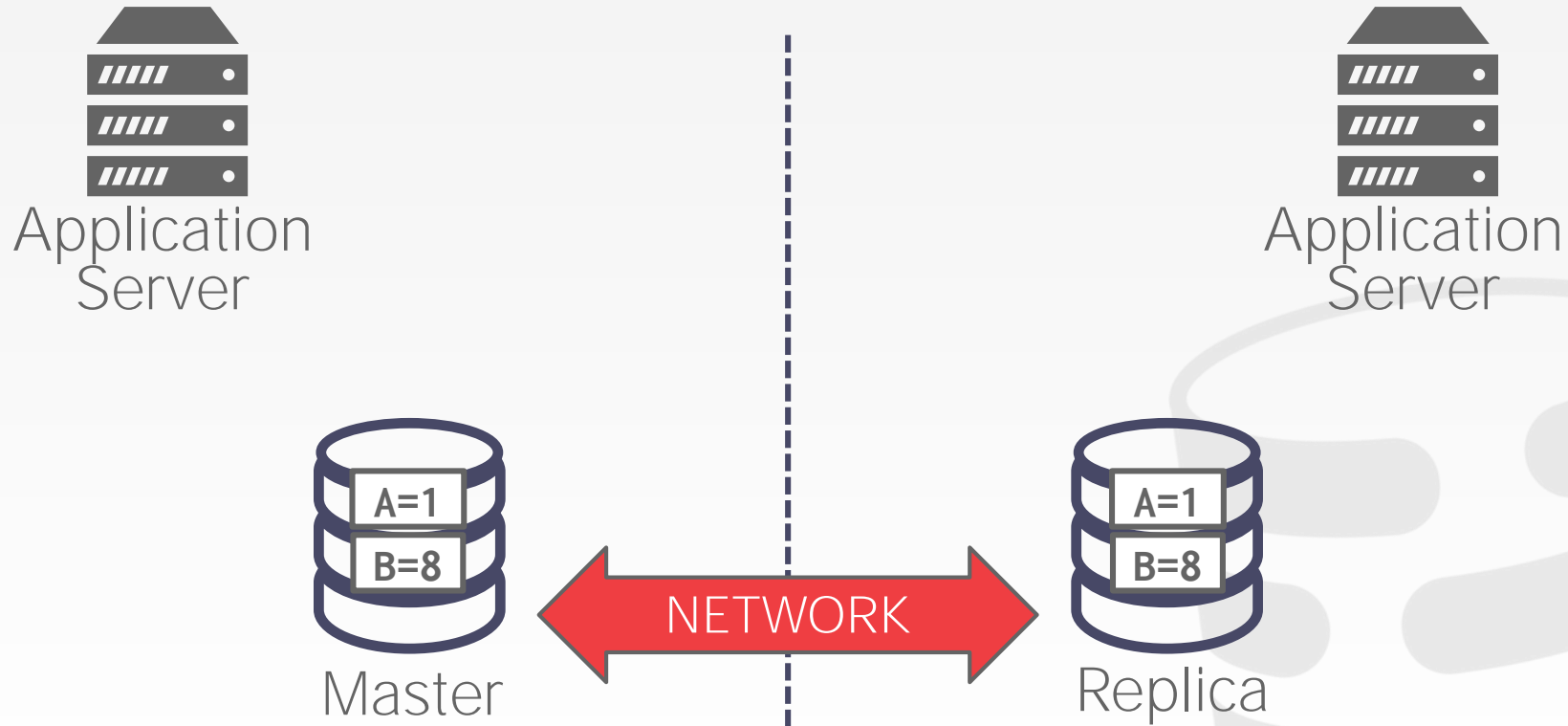
# CAP – AVAILABILITY



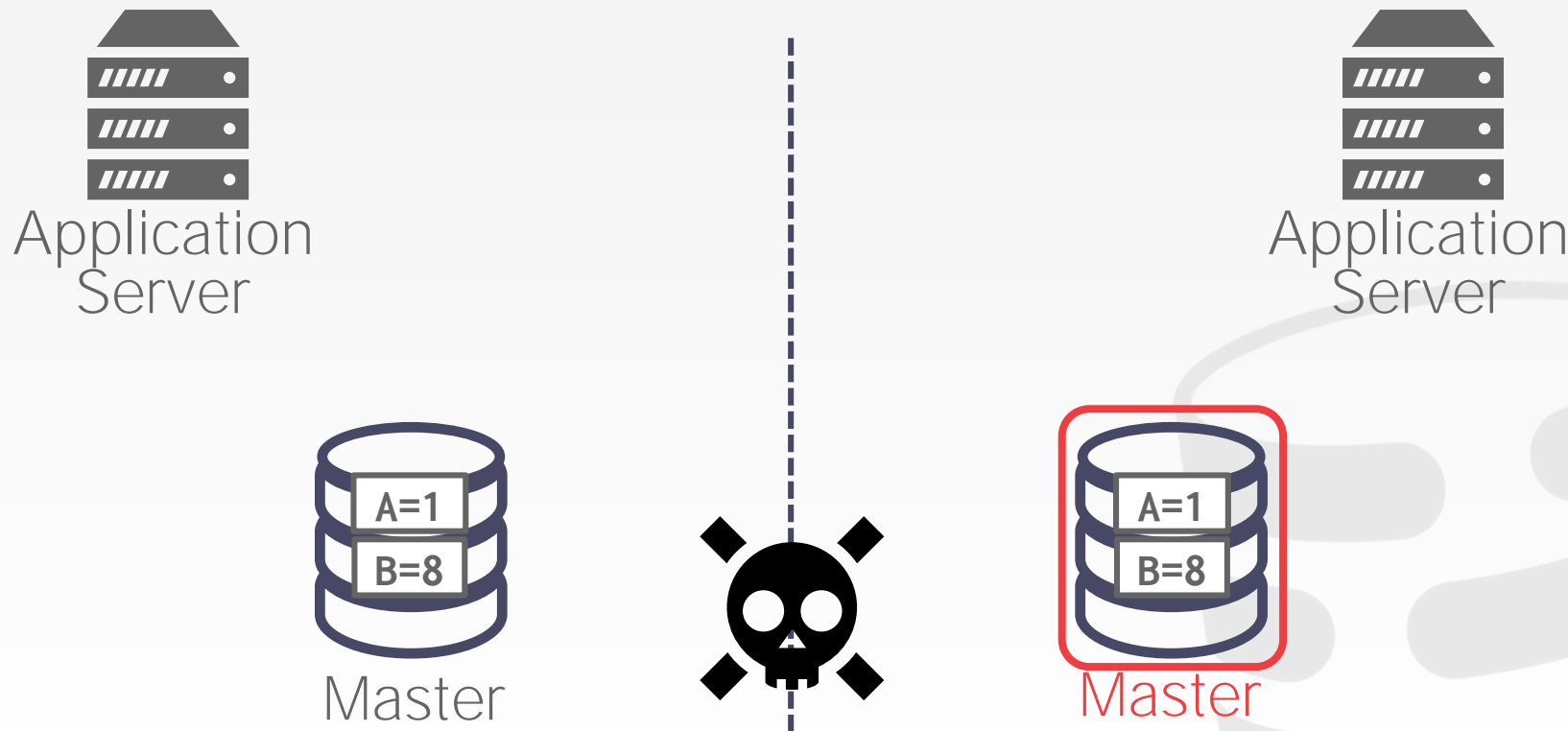
# CAP – AVAILABILITY



# CAP – PARTITION TOLERANCE

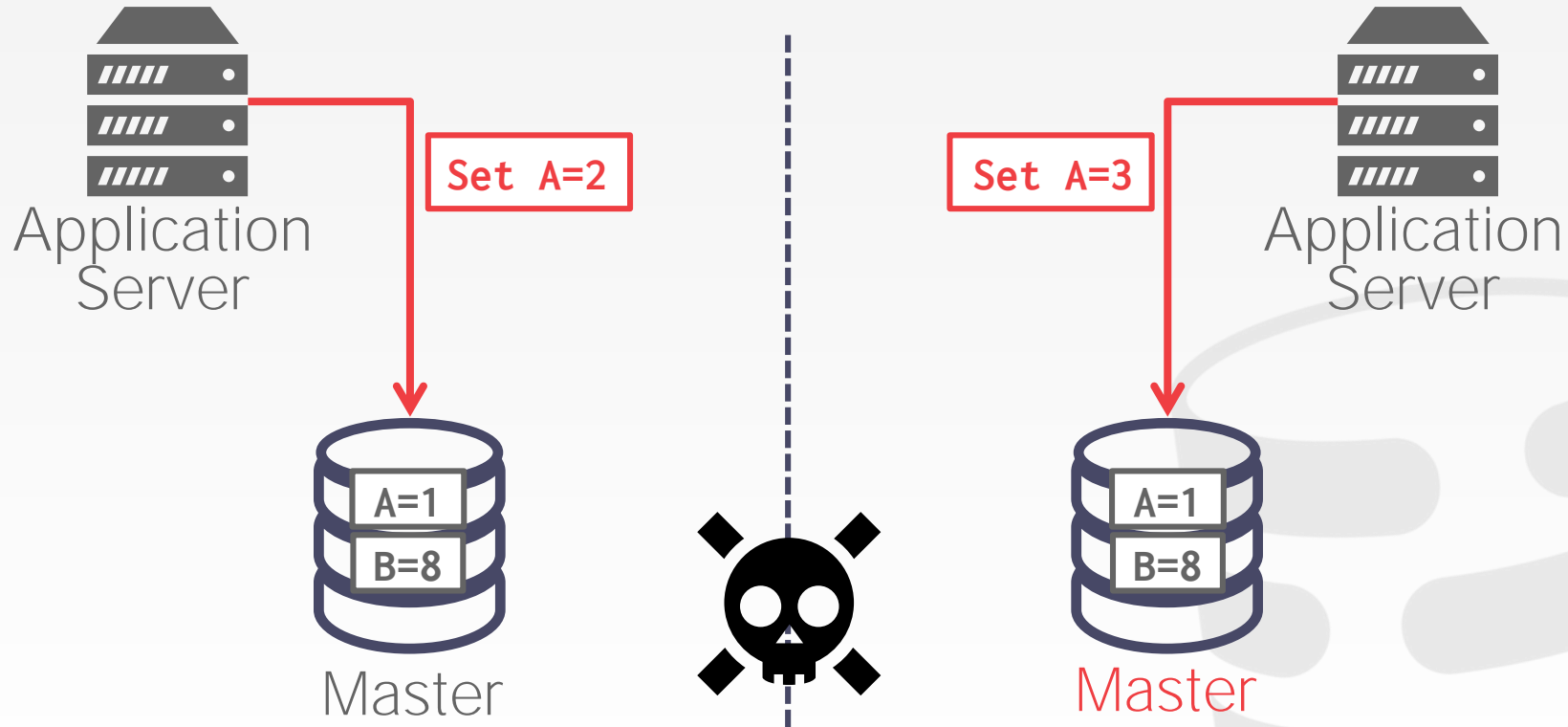


# CAP – PARTITION TOLERANCE

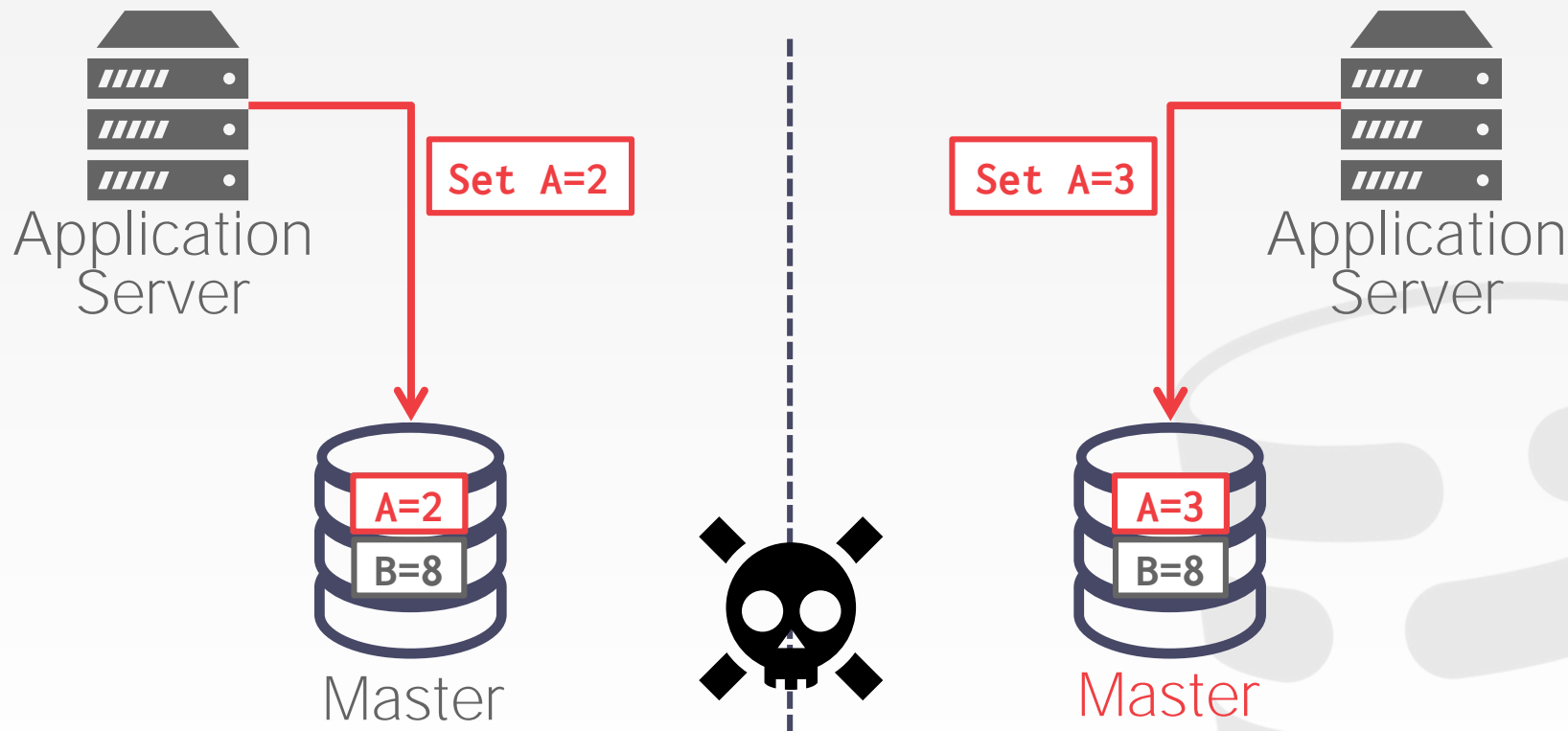




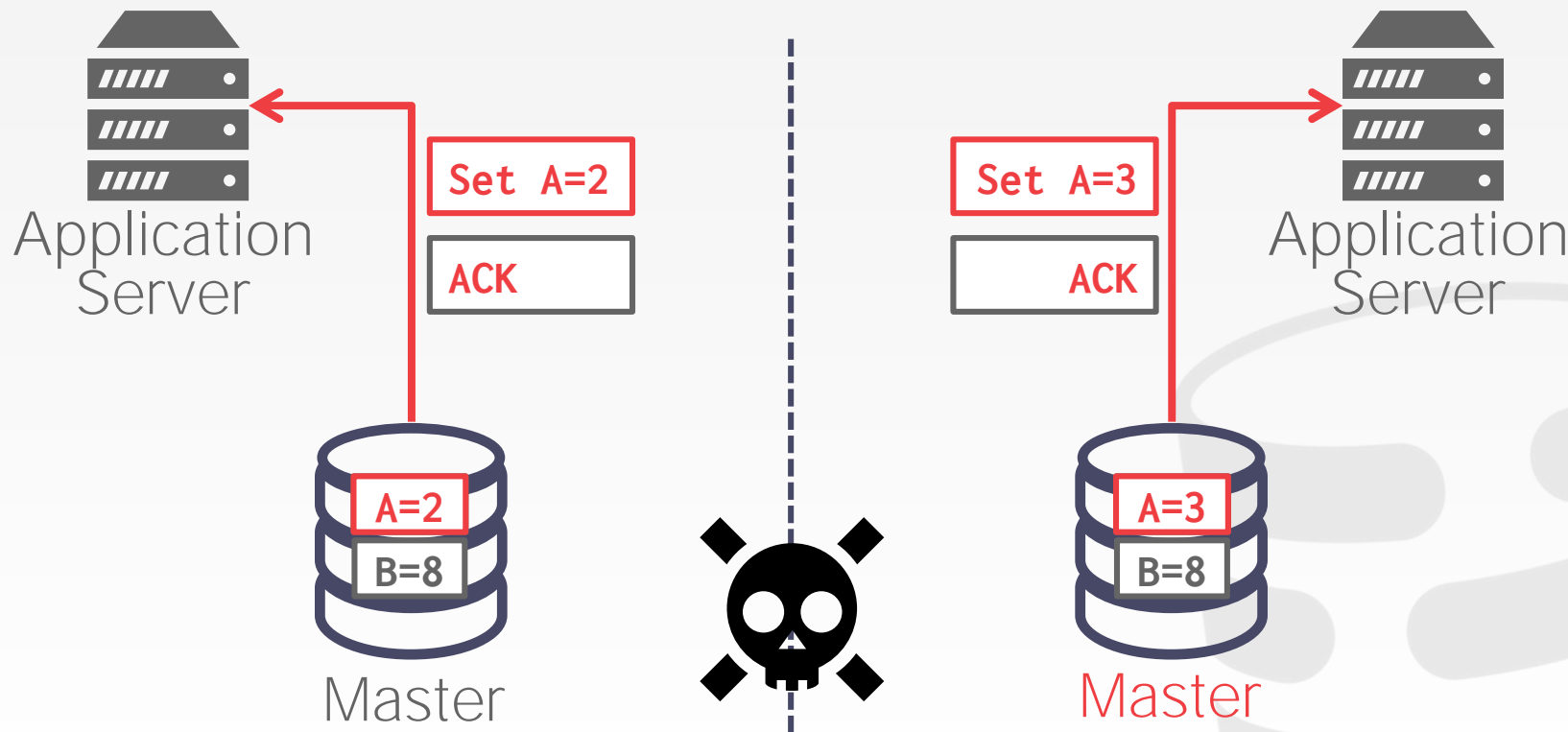
# CAP – PARTITION TOLERANCE



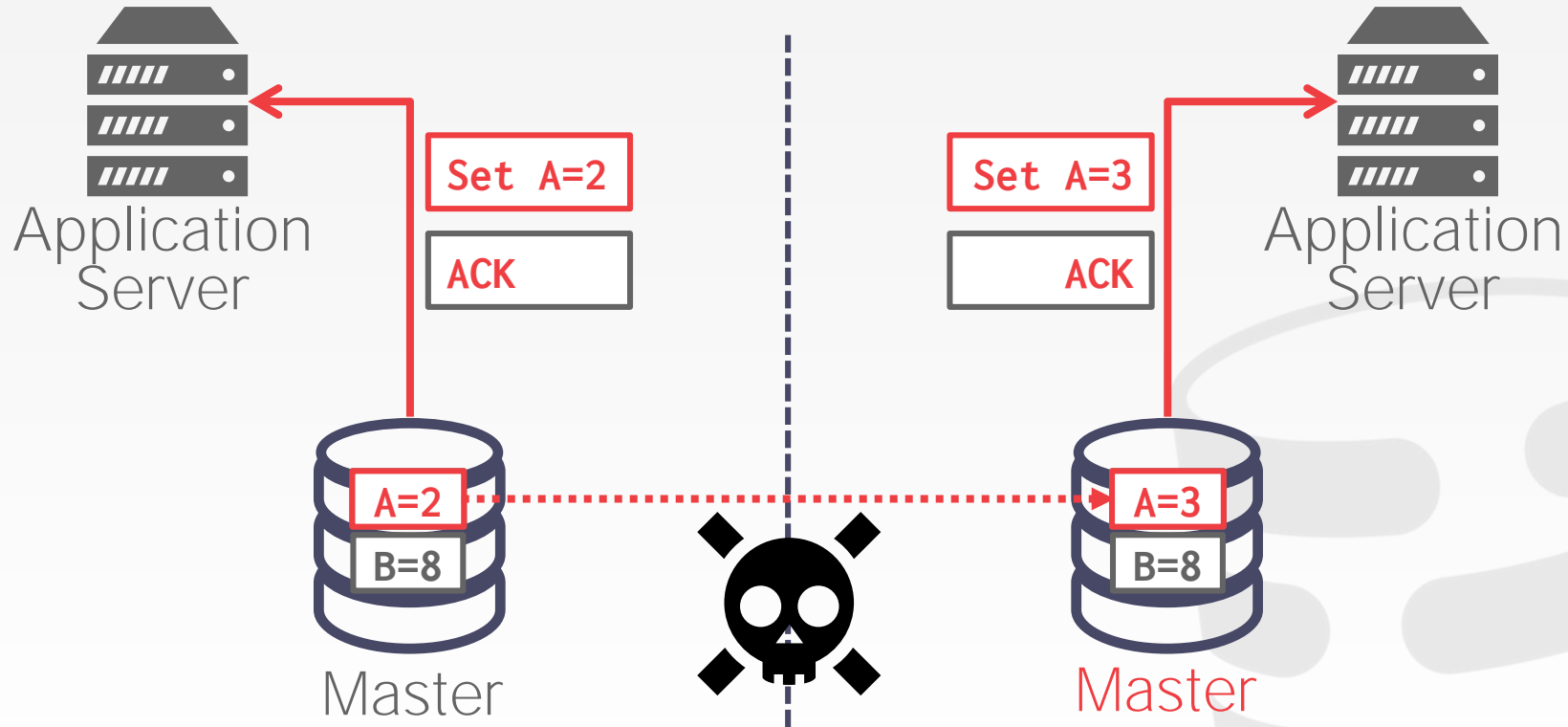
# CAP – PARTITION TOLERANCE



# CAP – PARTITION TOLERANCE



# CAP – PARTITION TOLERANCE



# CAP FOR OLTP DBMSs

---

How a DBMS handles failures determines which elements of the CAP theorem they support.

## **Traditional/NewSQL DBMSs**

→ Stop allowing updates until a majority of nodes are reconnected.

## **NoSQL DBMSs**

→ Provide mechanisms to resolve conflicts after nodes are reconnected.

# OBSERVATION

---

We have assumed that the nodes in our distributed systems are running the same DBMS software.

But organizations often run many different DBMSs in their applications.

It would be nice if we could have a single interface for all our data.



# FEDERATED DATABASES

---

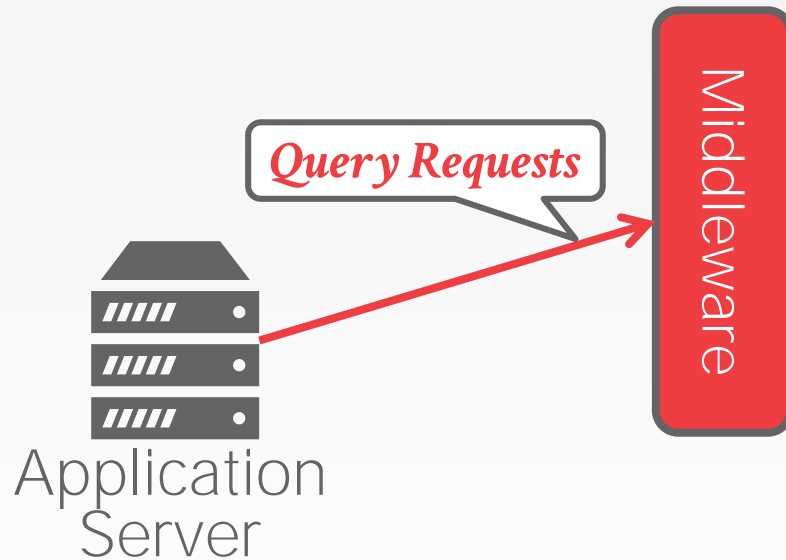
Distributed architecture that connects together multiple DBMSs into a single logical system.

A query can access data at any location.

This is hard and nobody does it well

- Different data models, query languages, limitations.
- No easy way to optimize queries
- Lots of data copying (bad).

# FEDERATED DATABASE EXAMPLE

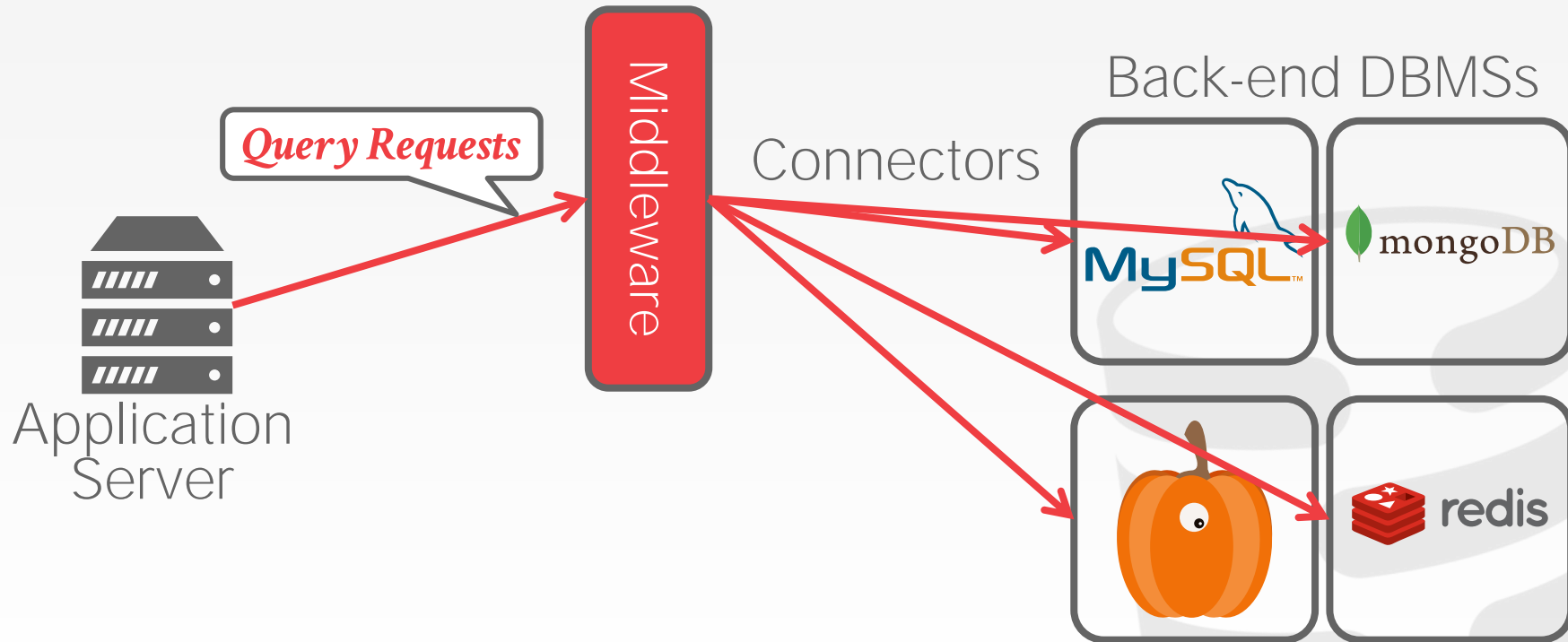


## Back-end DBMSs

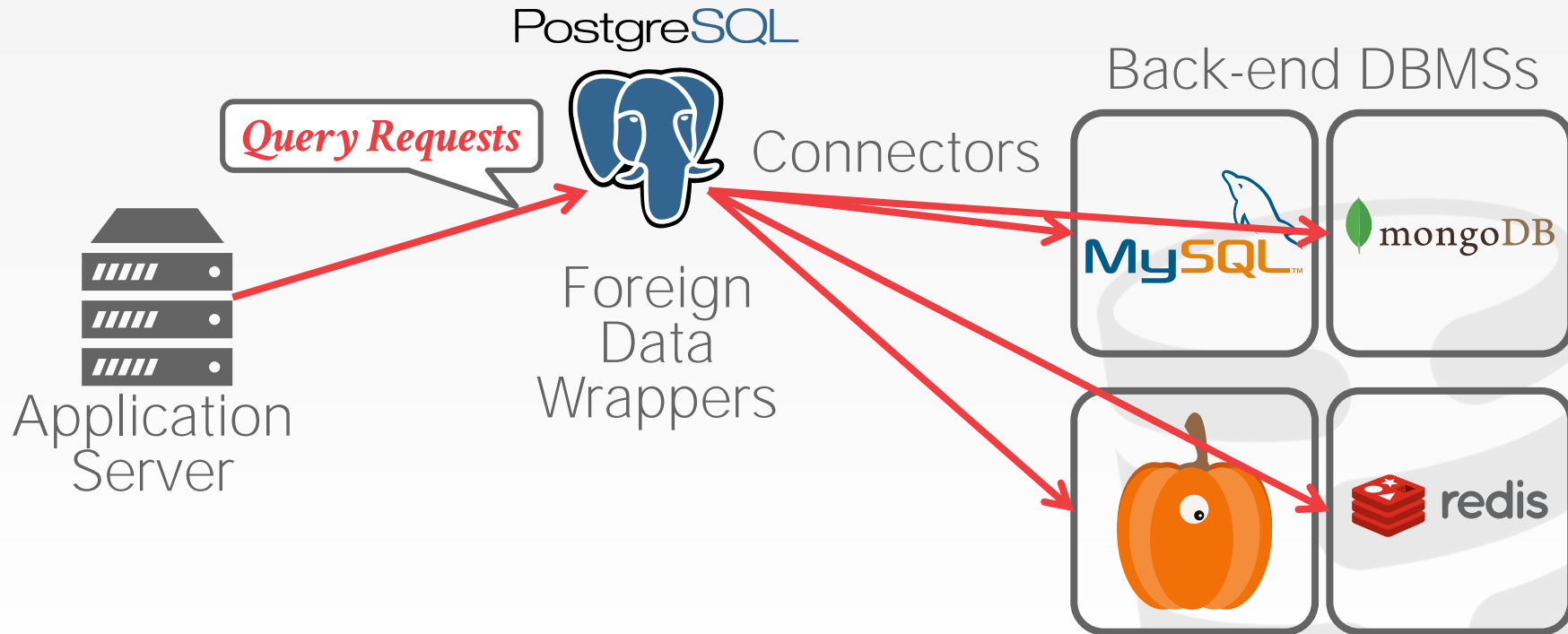




# FEDERATED DATABASE EXAMPLE



# FEDERATED DATABASE EXAMPLE



# CONCLUSION

---

We assumed that the nodes in our distributed DBMS are friendly.

Blockchain databases assume that the nodes are adversarial. This means you have to use different protocols to commit transactions.



# NEXT CLASS

---

## Distributed OLAP Systems

