Intro to Database Systems (15-445/645)

22 Distributed OLTP Databases

Carnegie Mellon University

FALL 2022

Andy Pavlo

ADMINISTRIVIA

Homework #5 is due **Sunday Dec 4th @ 11:59pm**

Project #4 is due **Sunday Dec 11th @ 11:59pm**

Upcoming Special Lectures:
→ **Snowflake** (Tuesday Dec 6th)
→ **Live Call-in Q&A Lecture** (Thursday Dec 8th)

Final Exam is **Friday Dec 16th @ 1:00pm.**

LAST CLASS

System Architectures
→ Shared-Memory, Shared-Disk, Shared-Nothing

Partitioning/Sharding
→ Hash, Range, Round Robin

Transaction Coordination
→ Centralized vs. Decentralized

OLTP VS. OLAP

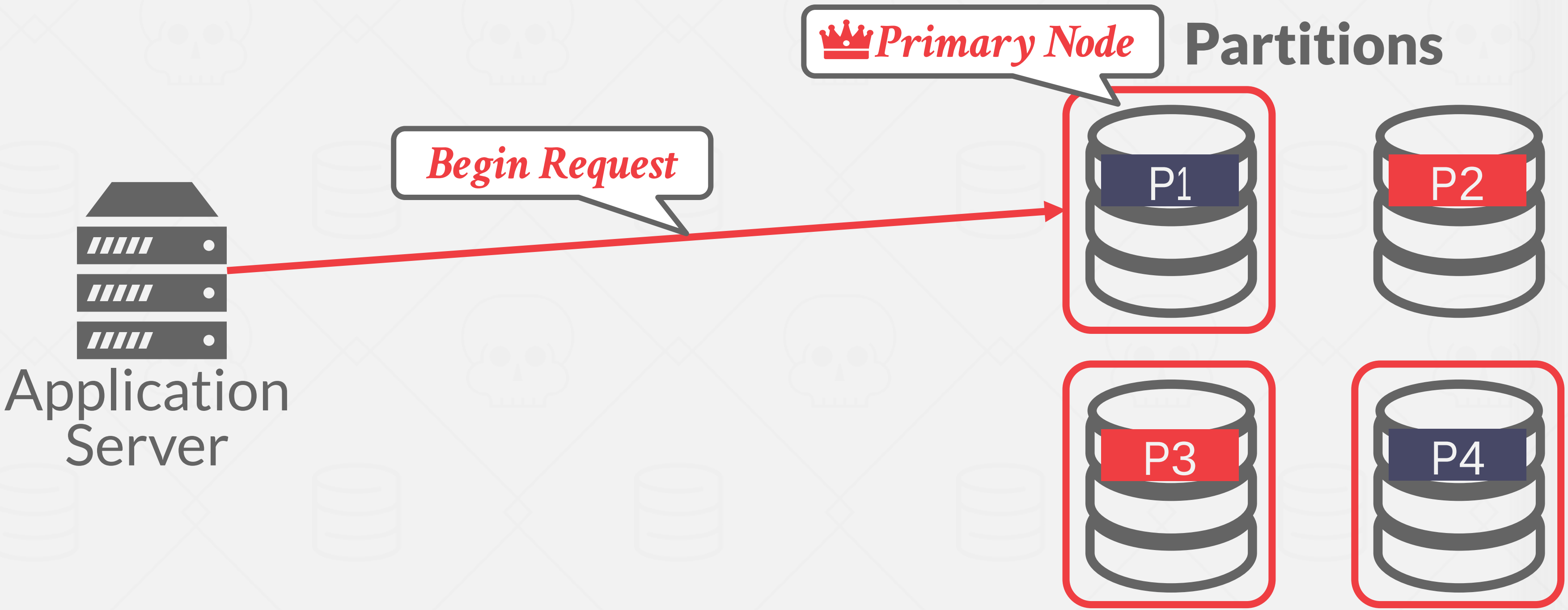
On-line Transaction Processing (OLTP):

→ Short-lived read/write txns.
→ Small footprint.
→ Repetitive operations.

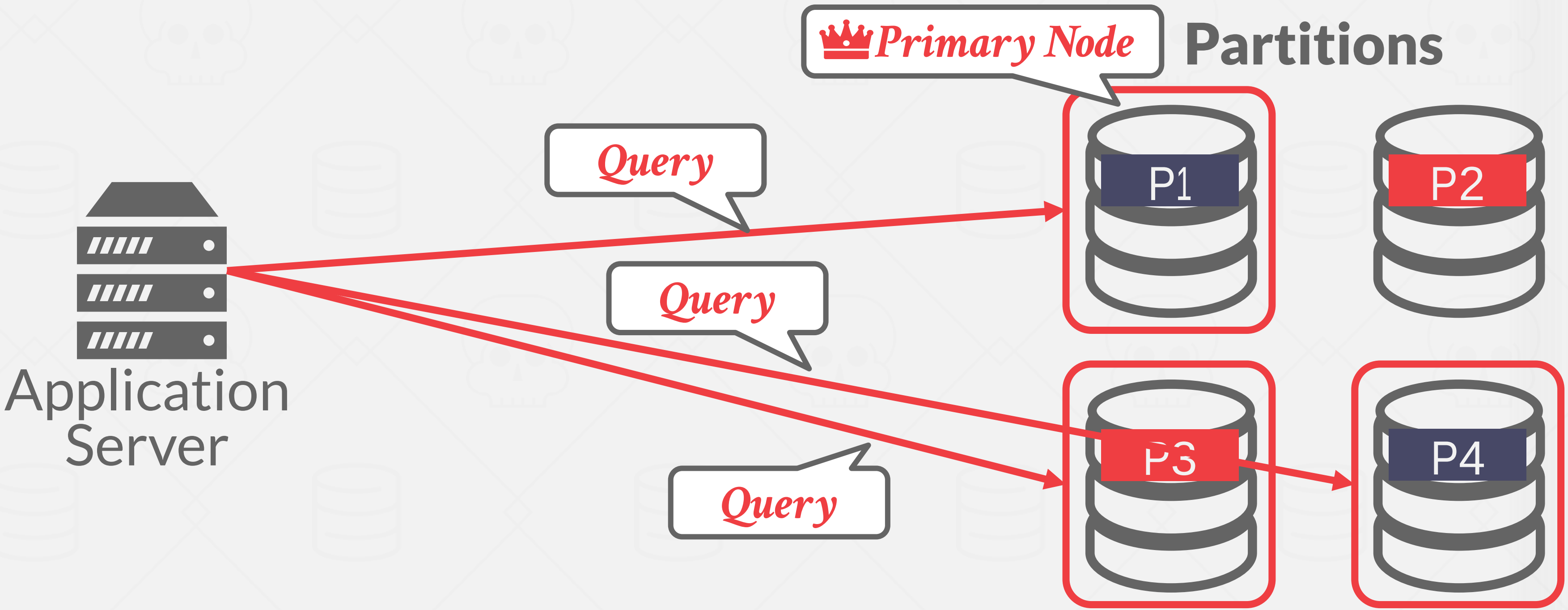
On-line Analytical Processing (OLAP):

→ Long-running, read-only queries.
→ Complex joins.
→ Exploratory queries.

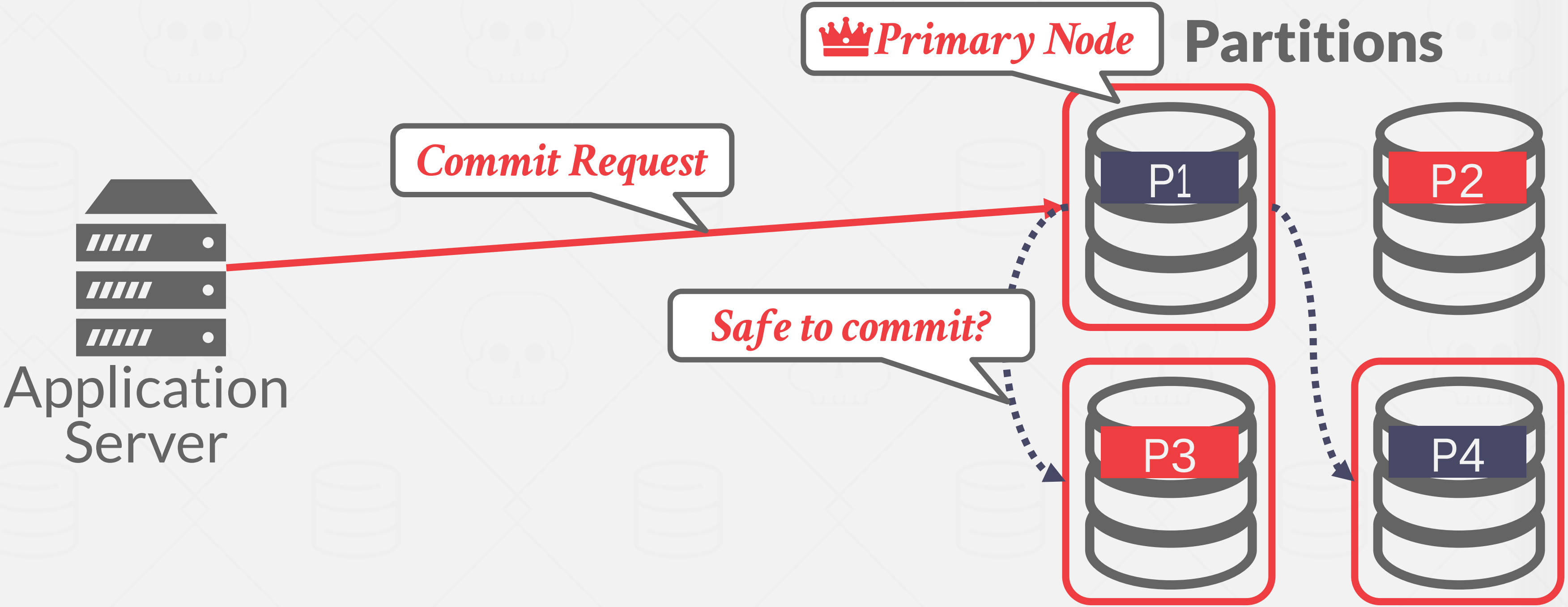
DECENTRALIZED COORDINATOR



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?

IMPORTANT ASSUMPTION

We will assume that all nodes in a distributed DBMS are well-behaved and under the same administrative domain.

→ If we tell a node to commit a txn, then it will commit the txn (if there is not a failure).

If you do not trust the other nodes in a distributed DBMS, then you need to use a Byzantine Fault Tolerant protocol for txns (blockchain).

TODAY'S AGENDA

Atomic Commit Protocols

Replication

Consistency Issues (CAP / PACELC)

Google Spanner

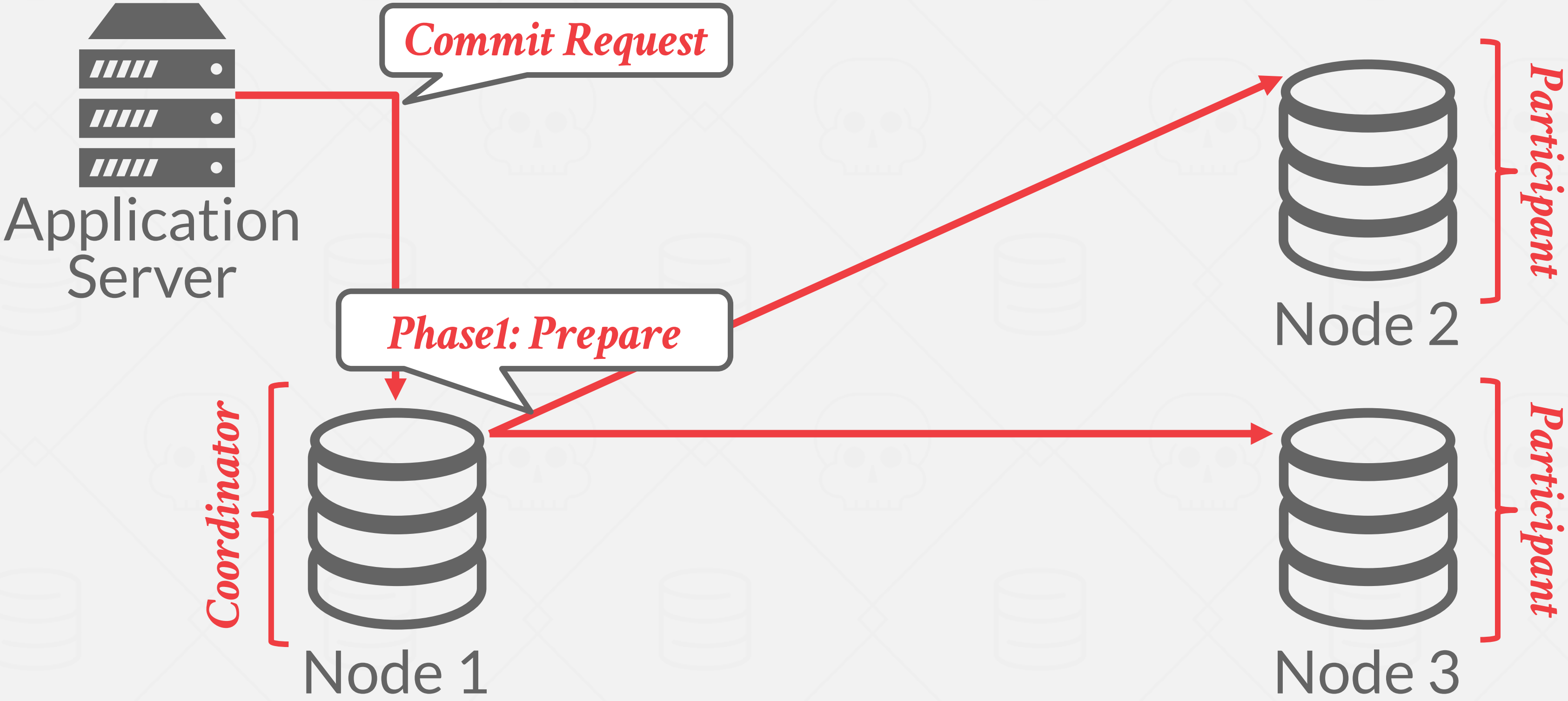
ATOMIC COMMIT PROTOCOL

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

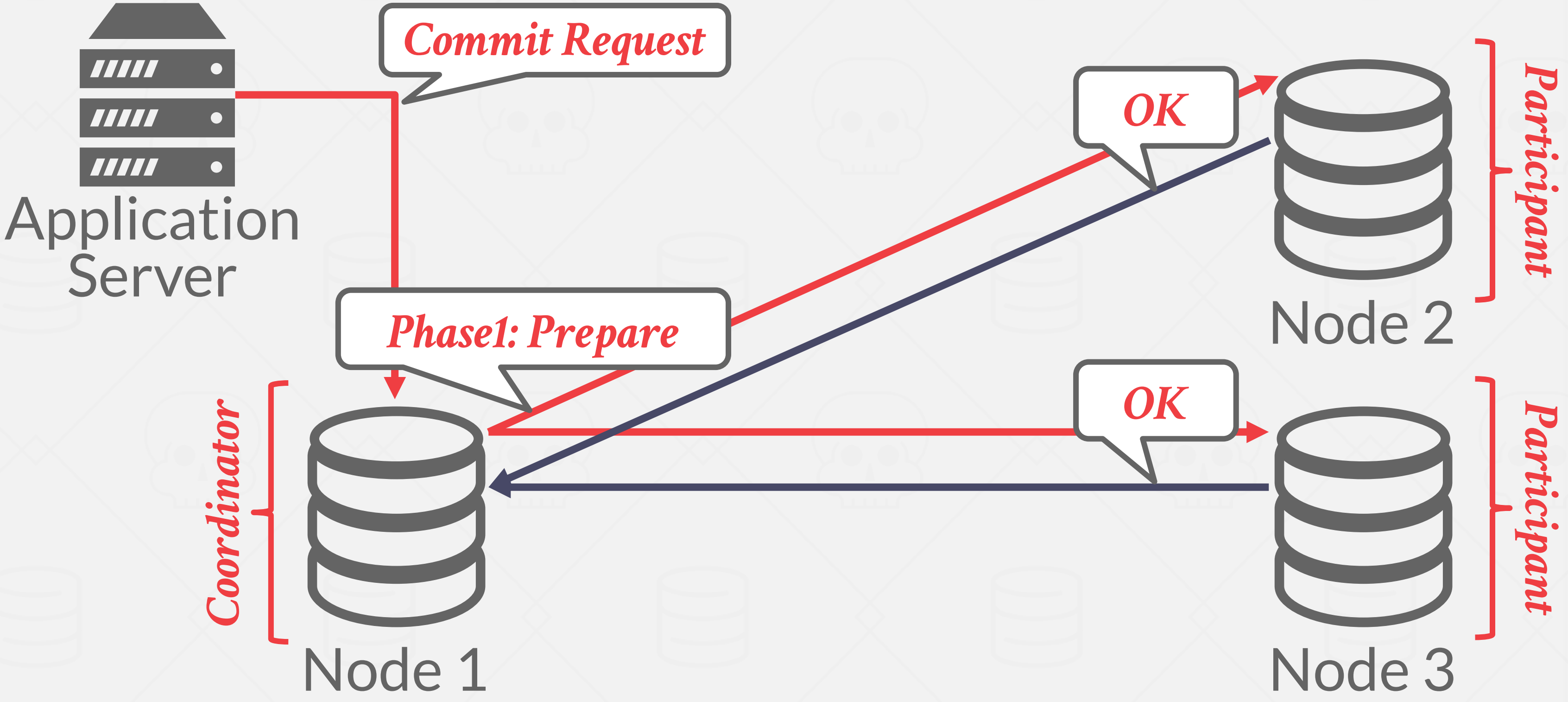
Examples:
→ Two-Phase Commit
→ Three-Phase Commit (not used)
→ Paxos
→ Raft
→ ZAB (Apache Zookeeper)
→ Viewstamped Replication

TWO-PHASE COMMIT (SUCCESS)

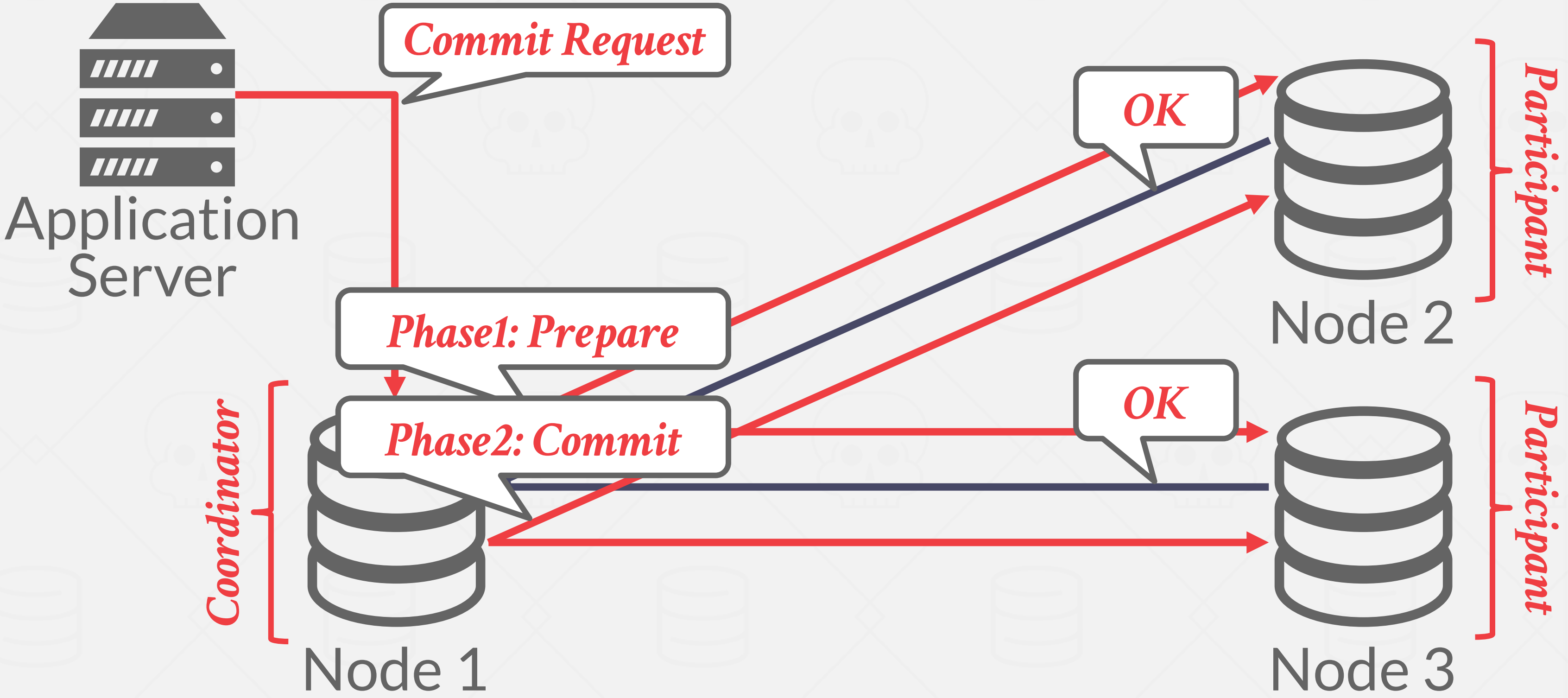
TWO-PHASE COMMIT (SUCCESS)



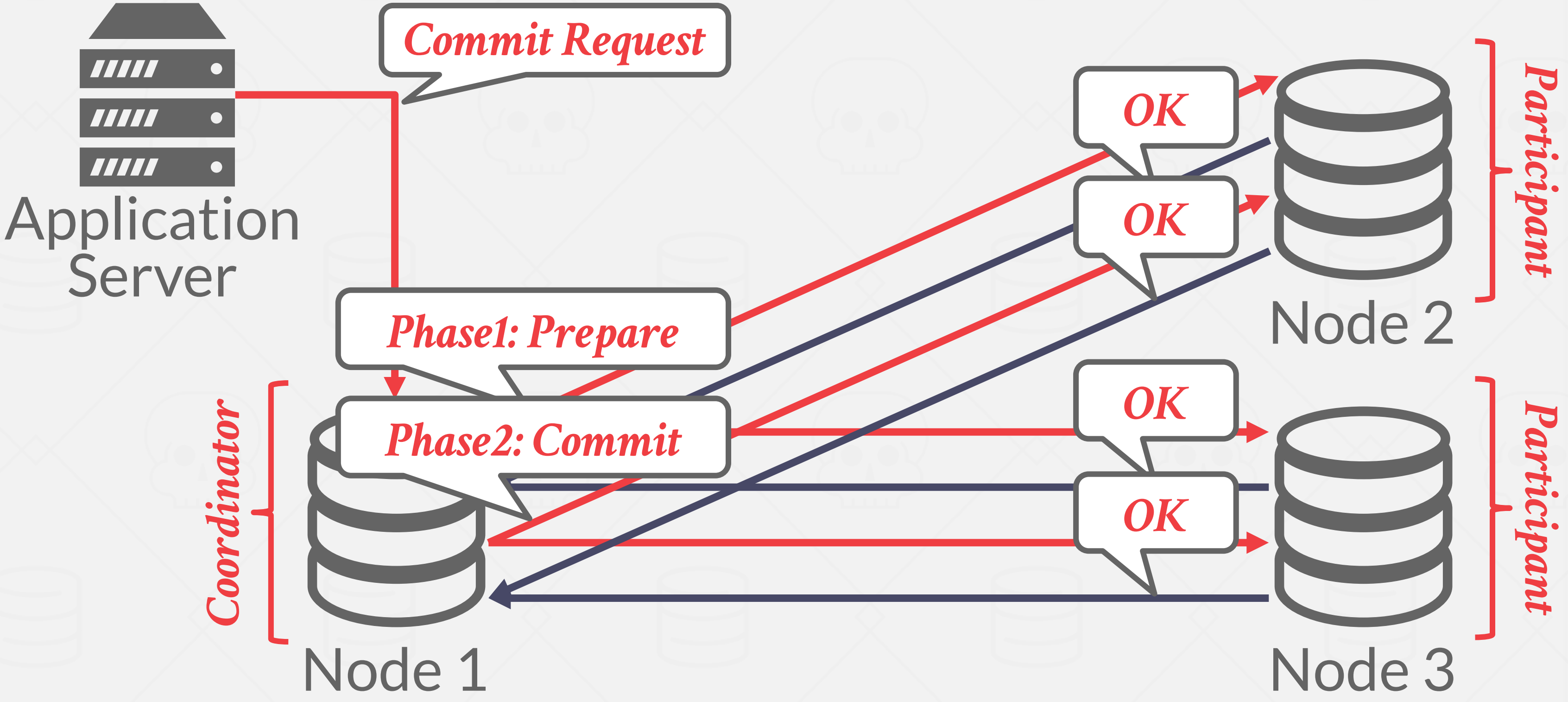
TWO-PHASE COMMIT (SUCCESS)



TWO-PHASE COMMIT (SUCCESS)

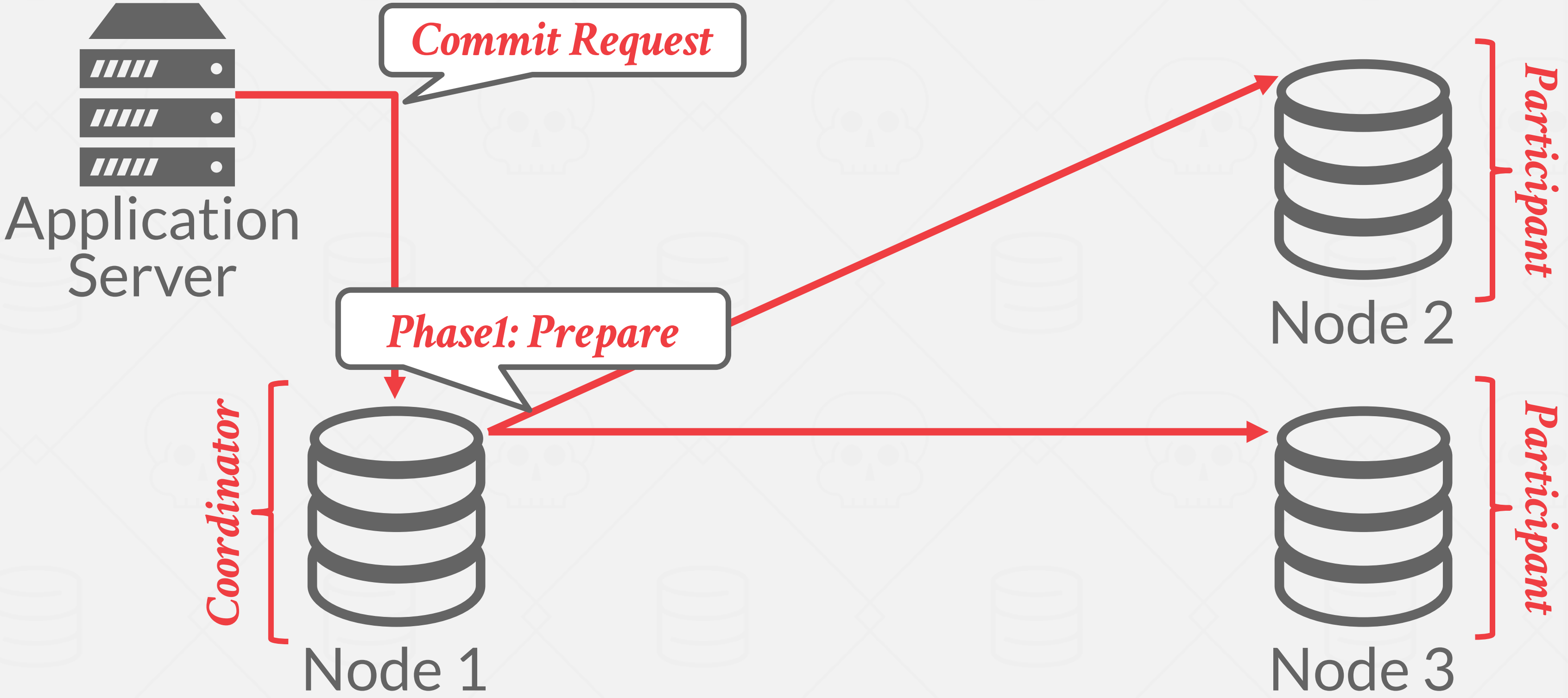


TWO-PHASE COMMIT (SUCCESS)

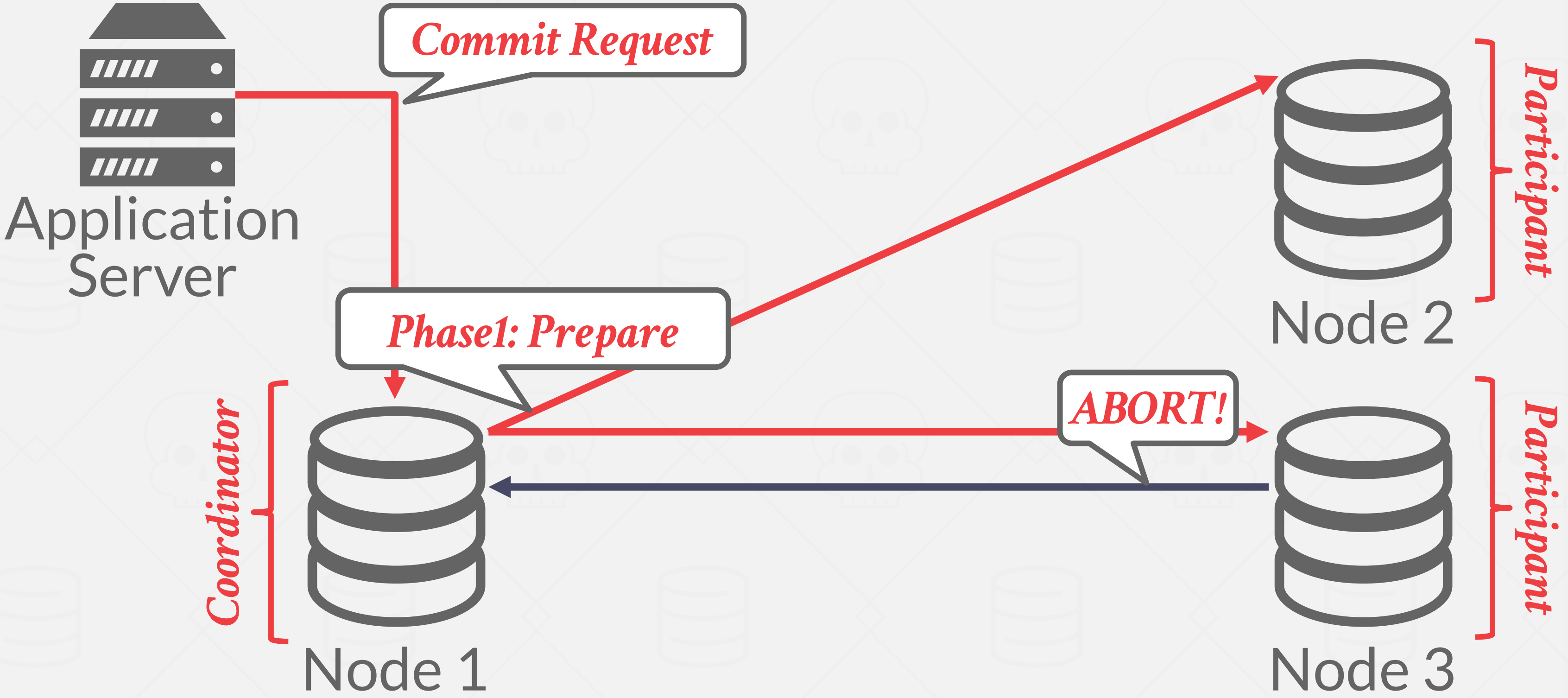


TWO-PHASE COMMIT (SUCCESS)

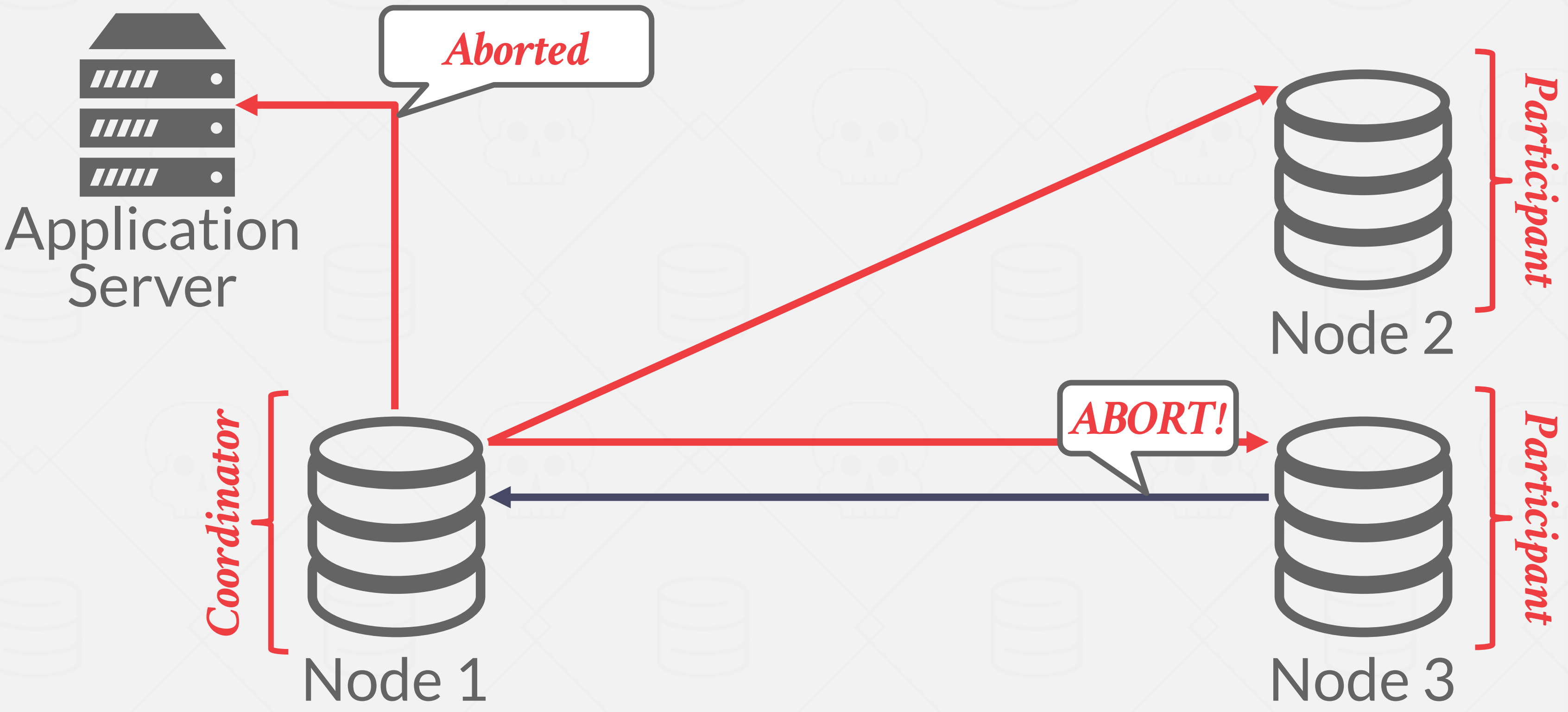
TWO-PHASE COMMIT (ABORT)



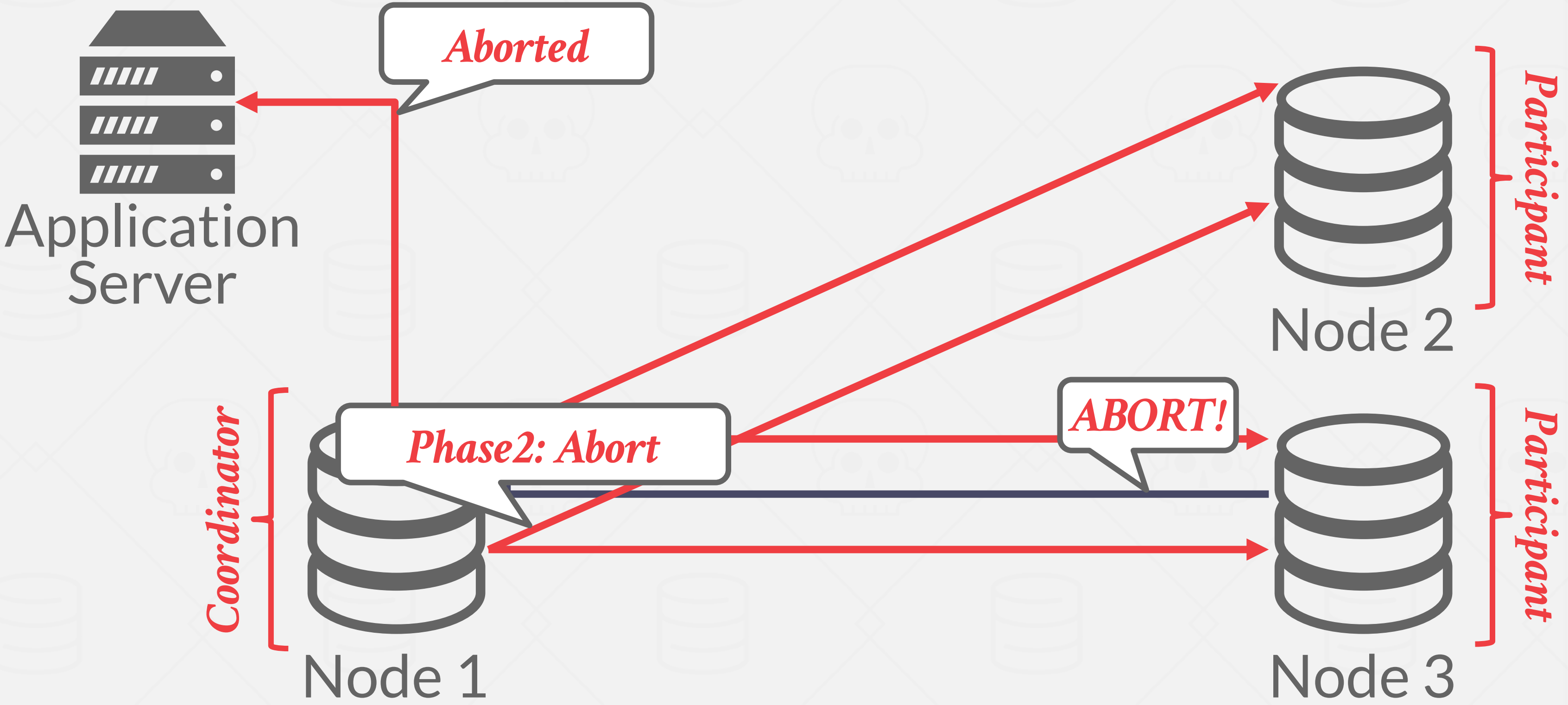
TWO-PHASE COMMIT (ABORT)



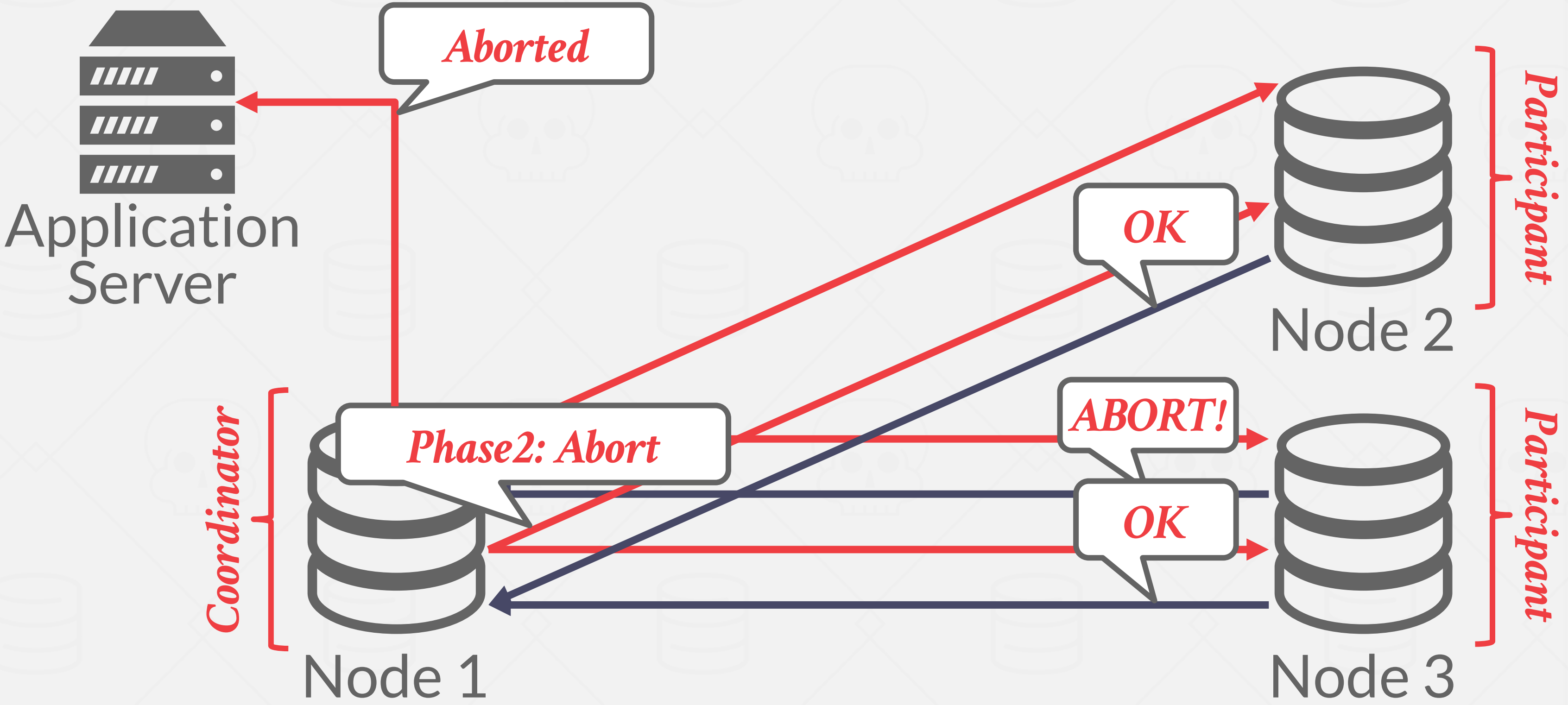
TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT (ABORT)



TWO-PHASE COMMIT

Each node records the inbound/outbound messages and outcome of each phase in a non-volatile storage log.

On recovery, examine the log for 2PC messages:
→ If local txn in prepared state, contact coordinator.
→ If local txn not in prepared, abort it.
→ If local txn was committing and node is the coordinator, send COMMIT message to nodes.

TWO-PHASE COMMIT FAILURES

What happens if coordinator crashes?

→ Participants must decide what to do after a timeout.
→ System is not available during this time.

What happens if participant crashes?

→ Coordinator assumes that it responded with an abort if it hasn't sent an acknowledgement yet.
→ Again, nodes use a timeout to determine that participant is dead.

2PC OPTIMIZATIONS

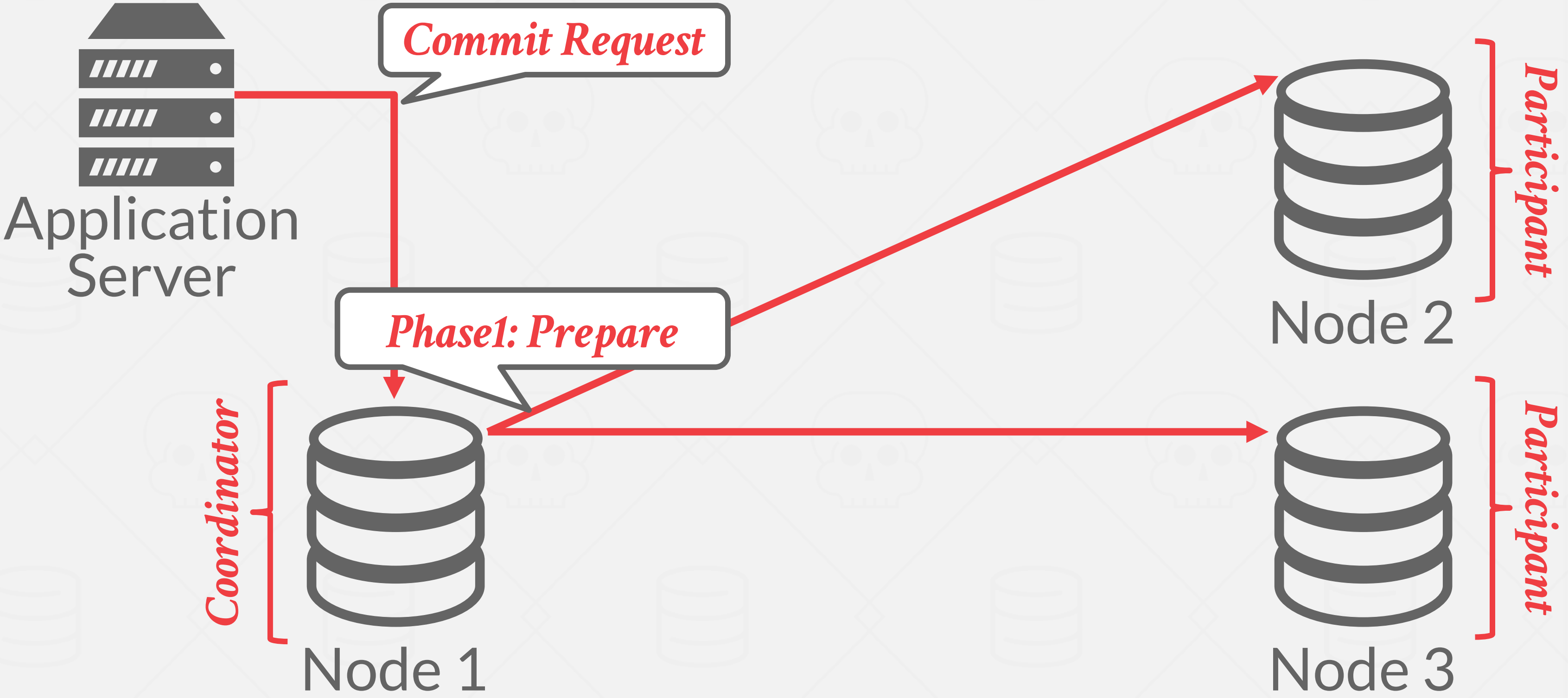
Early Prepare Voting *(Rare)*

→ 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.

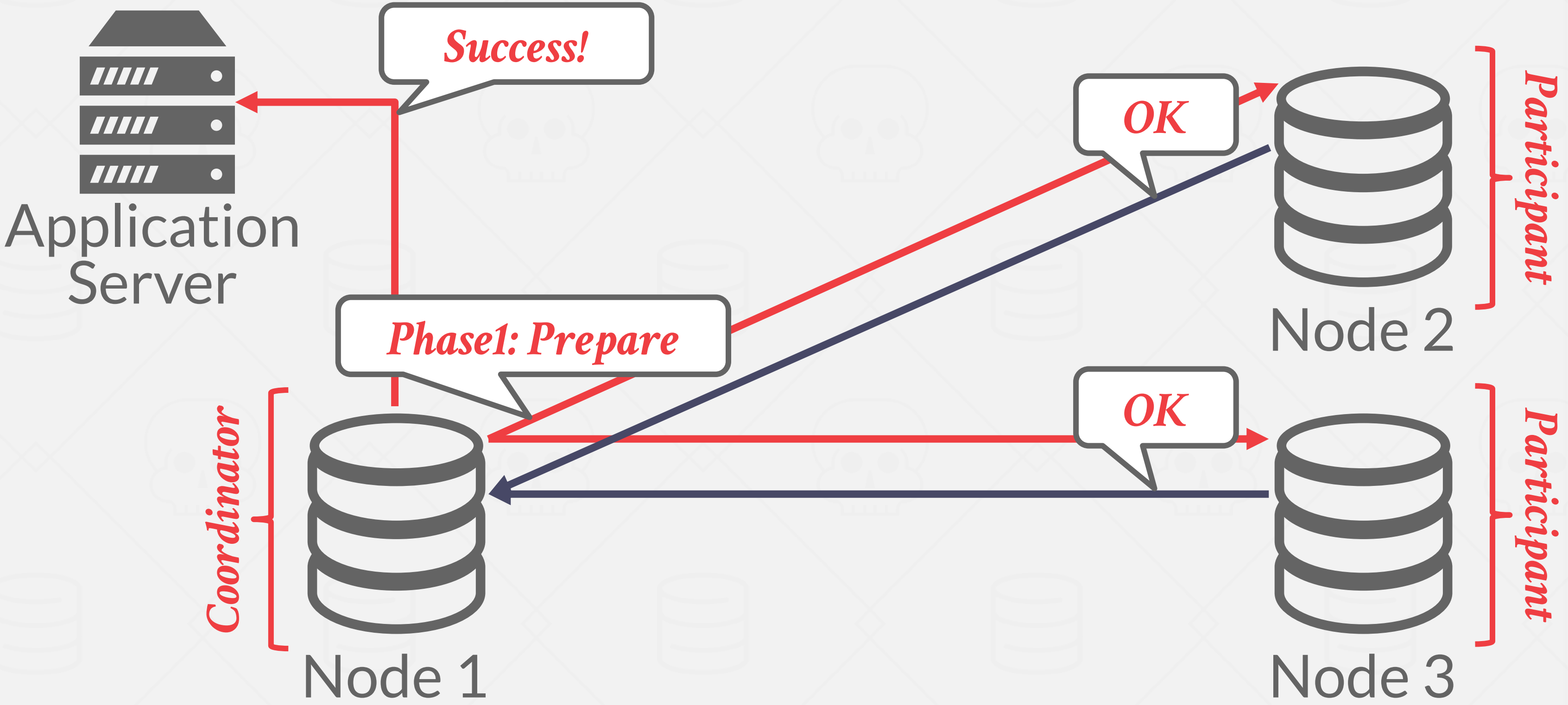
Early Ack After Prepare *(Common)*

→ 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.

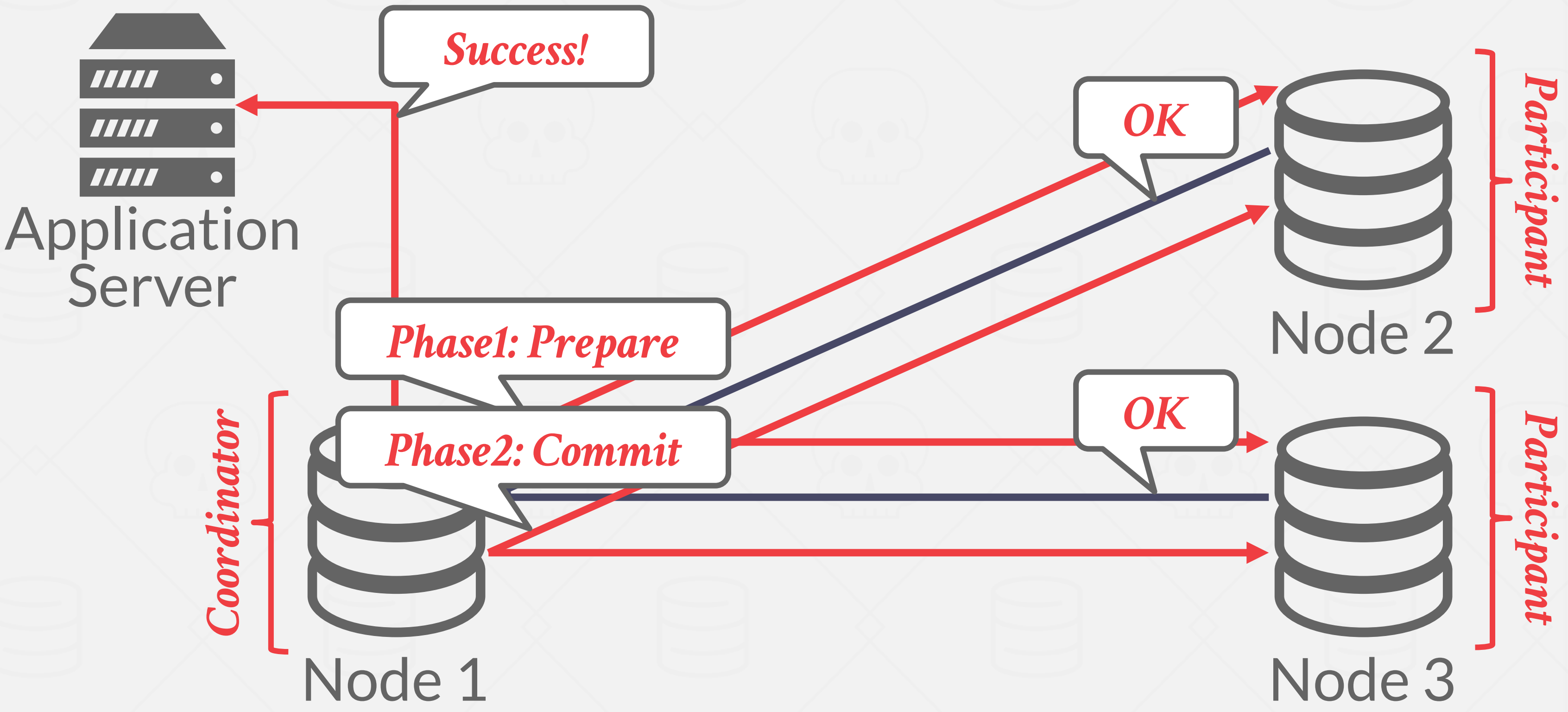
EARLY ACKNOWLEDGEMENT



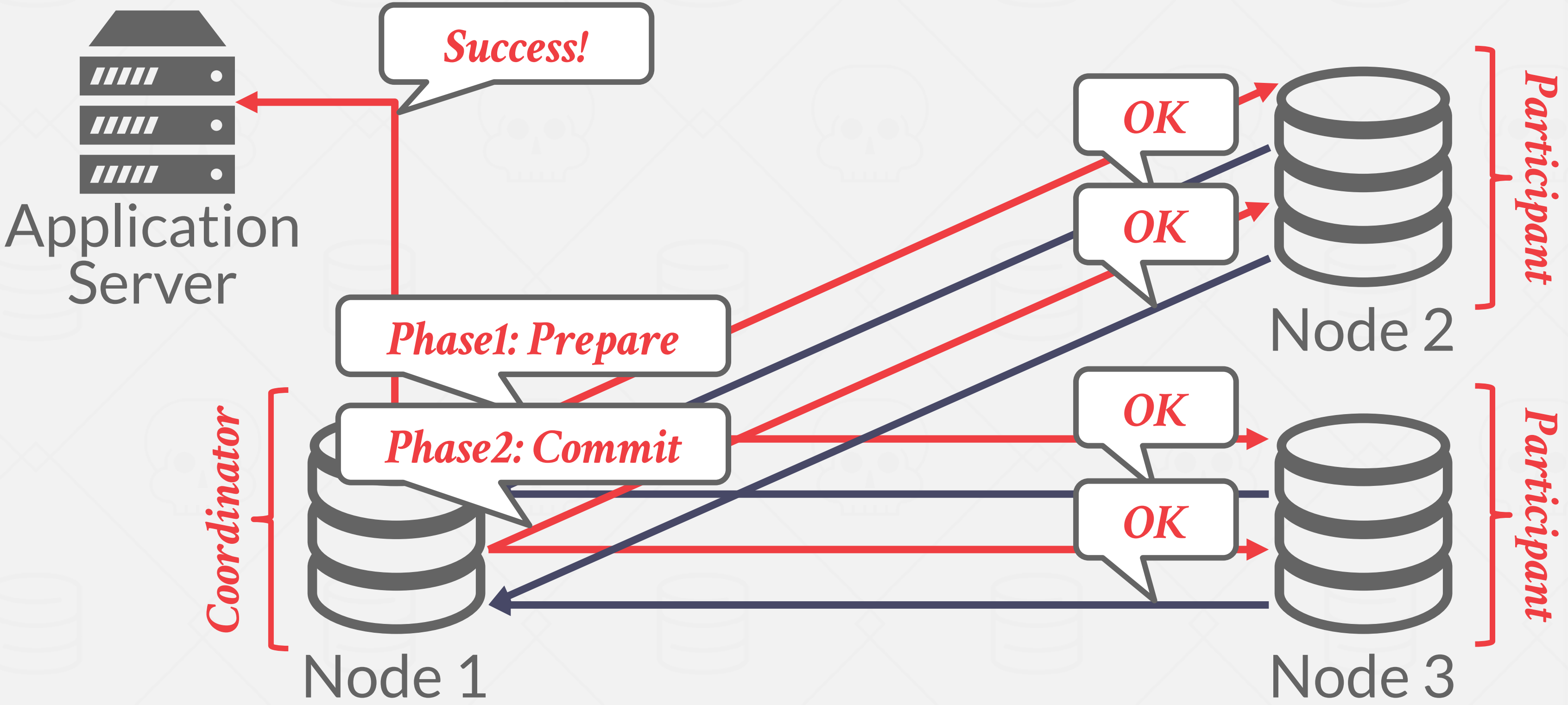
EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



EARLY ACKNOWLEDGEMENT



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 Paxon 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 Paxon 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 Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon 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 Lampson [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.

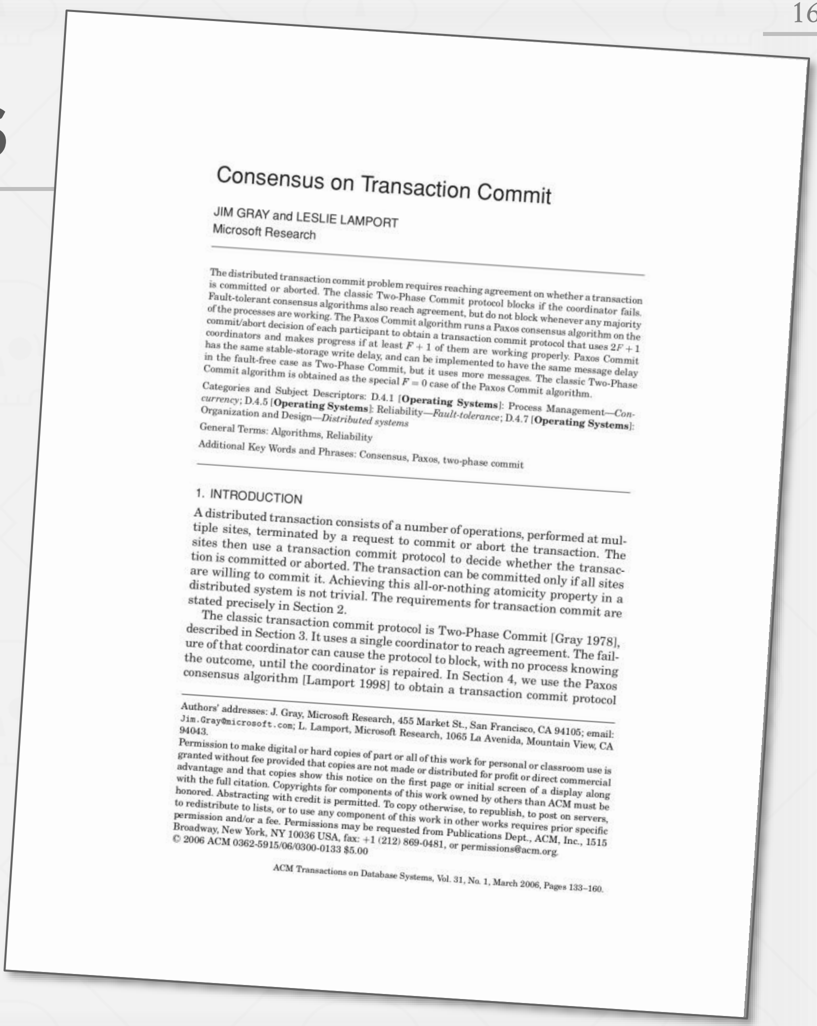
Keith Marzullo
University of California, San Diego

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, or to republish, requires a fee and/or specific permission.
© 1998 ACM 0000-0000/98/0000-0000 $00.00

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.



Consensus on Transaction Commit

JIM GRAY and LESLIE LAMPORT
Microsoft Research

The distributed transaction commit problem requires reaching agreement on whether a transaction is committed or aborted. The classic Two-Phase Commit protocol blocks if the coordinator fails. Fault-tolerant consensus algorithms also reach agreement, but do not block whenever any majority of the processes are working. The Paxos Commit algorithm runs a Paxos consensus algorithm on the commit/abort decision of each participant to obtain a transaction commit protocol that uses $2F+1$ coordinators and makes progress if at least $F+1$ of them are working properly. Paxos Commit has the same stable-storage write delay, and can be implemented to have the same message delay in the fault-free case as Two-Phase Commit, but it uses more messages. The classic Two-Phase Commit algorithm is obtained as the special $F = 0$ case of the Paxos Commit algorithm.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*Con-currency*; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*

General Terms: Algorithms, Reliability

Additional Key Words and Phrases: Consensus, Paxos, two-phase commit

1. INTRODUCTION

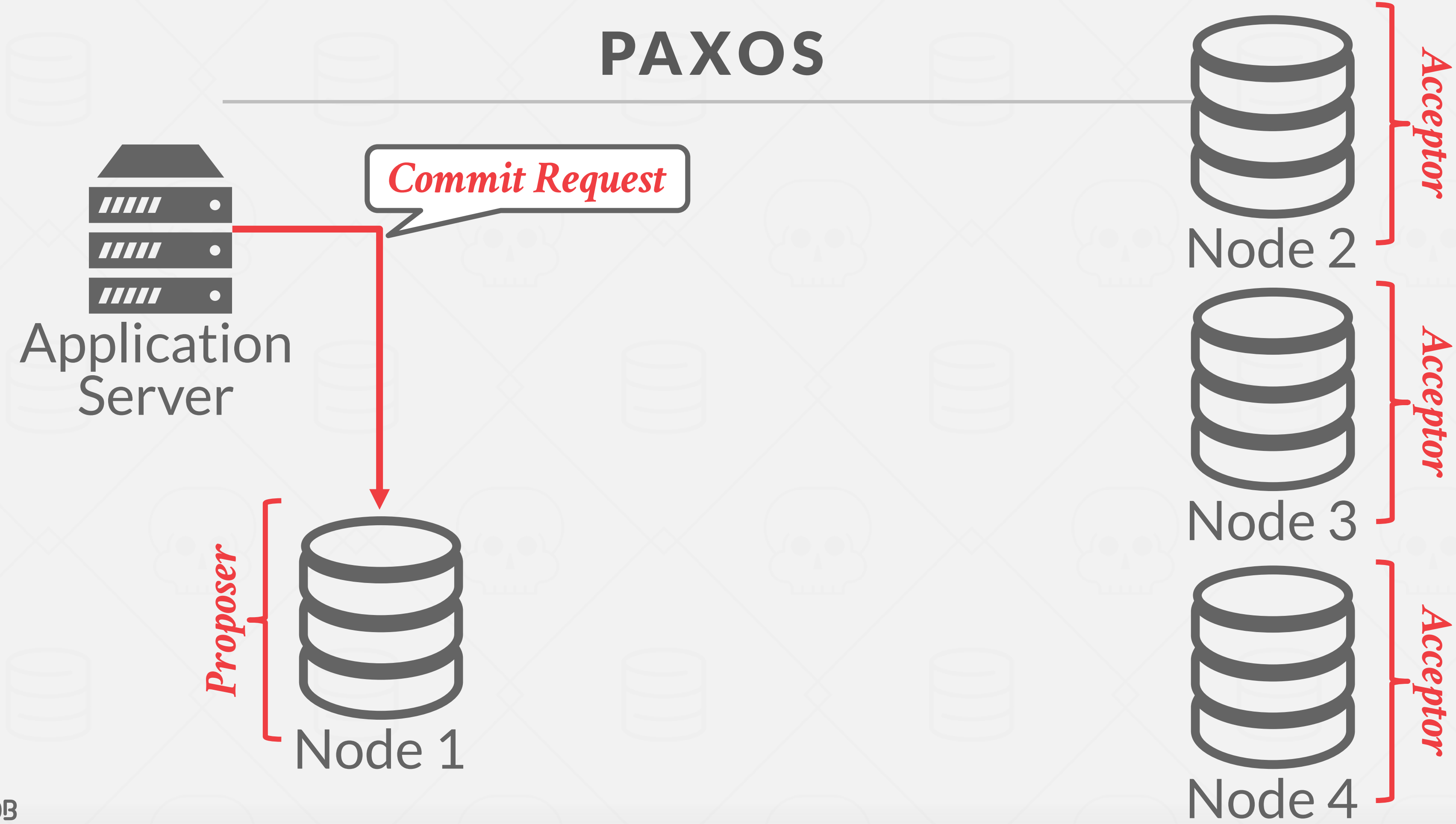
A distributed transaction consists of a number of operations, performed at multiple sites, terminated by a request to commit or abort the transaction. The sites then use a transaction commit protocol to decide whether the transaction is committed or aborted. The transaction can be committed only if all sites are willing to commit it. Achieving this all-or-nothing atomicity property in a distributed system is not trivial. The requirements for transaction commit are stated precisely in Section 2.

The classic transaction commit protocol is Two-Phase Commit [Gray 1978], described in Section 3. It uses a single coordinator to reach agreement. The failure of that coordinator can cause the protocol to block, with no process knowing the outcome, until the coordinator is repaired. In Section 4, we use the Paxos consensus algorithm [Lamport 1998] to obtain a transaction commit protocol

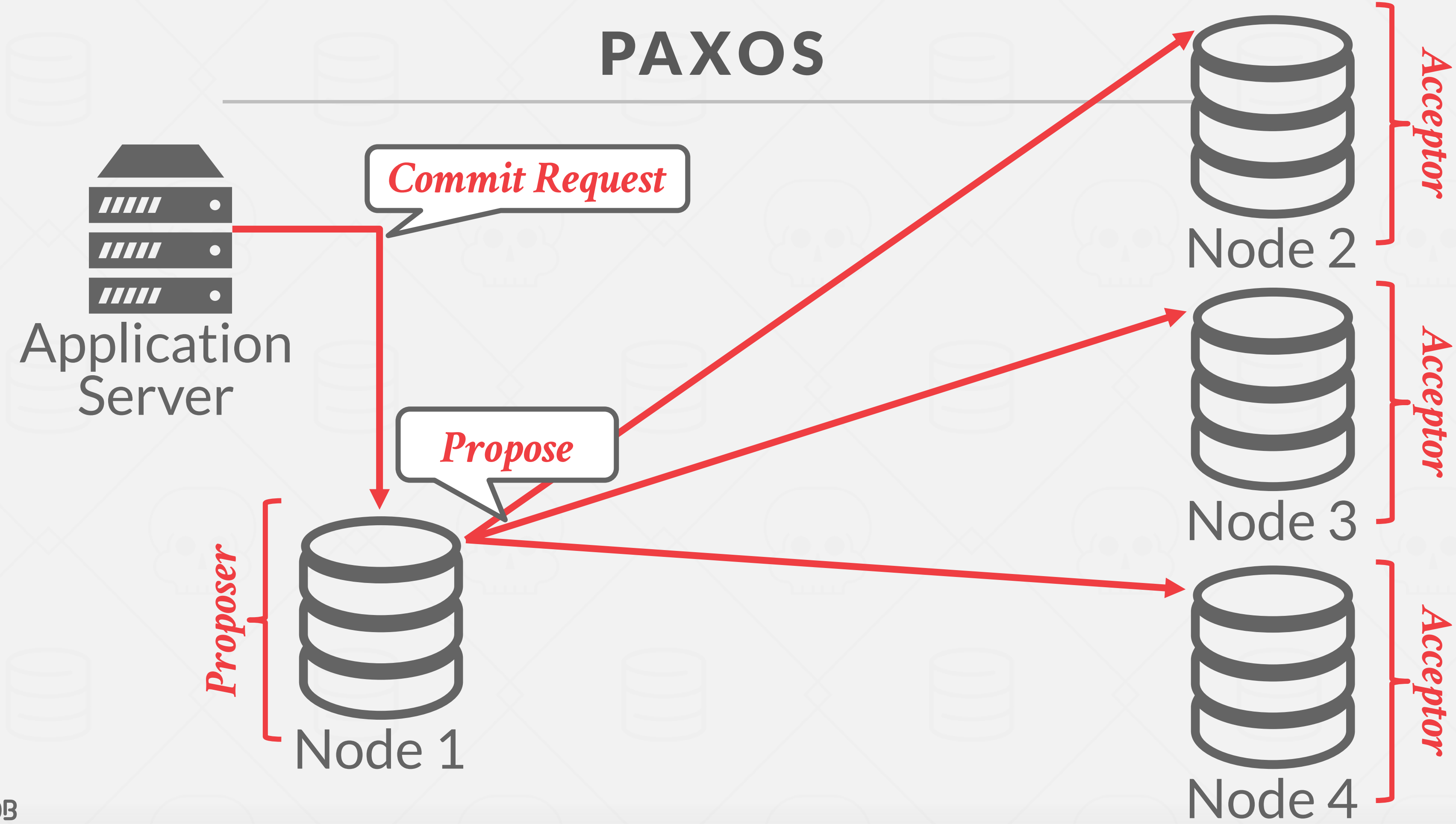
Authors' addresses: J. Gray, Microsoft Research, 455 Market St., San Francisco, CA 94105; email: Jim.Gray@microsoft.com; L. Lamport, Microsoft Research, 1065 La Avenida, Mountain View, CA 94043.

ACM Transactions on Database Systems, Vol. 31, No. 1, March 2006, Pages 133–160.

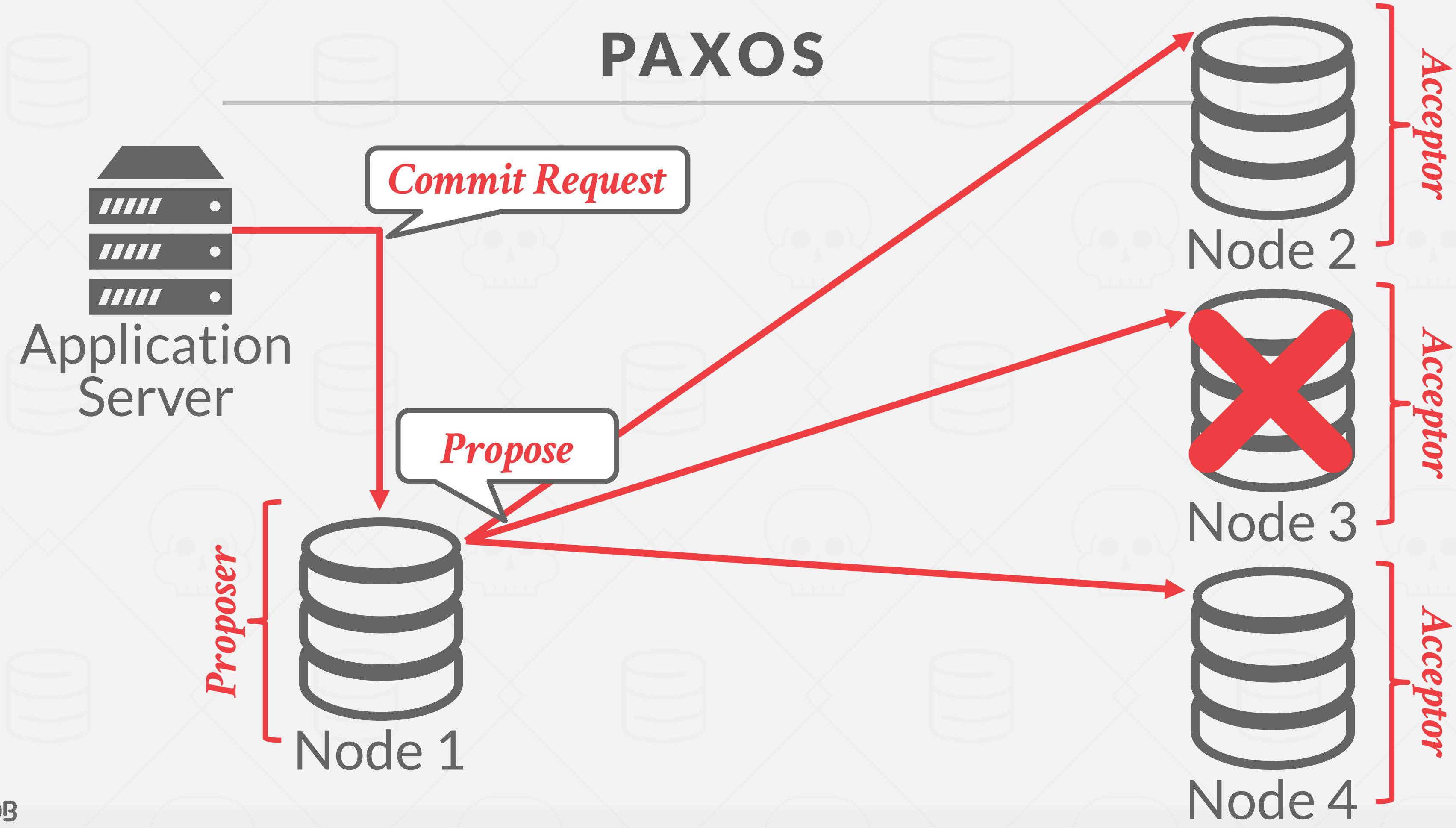
PAXOS


Commit Request
Application Server
Proposer
Node 1
Acceptor
Node 2
Acceptor
Node 3
Acceptor
Node 4

PAXOS

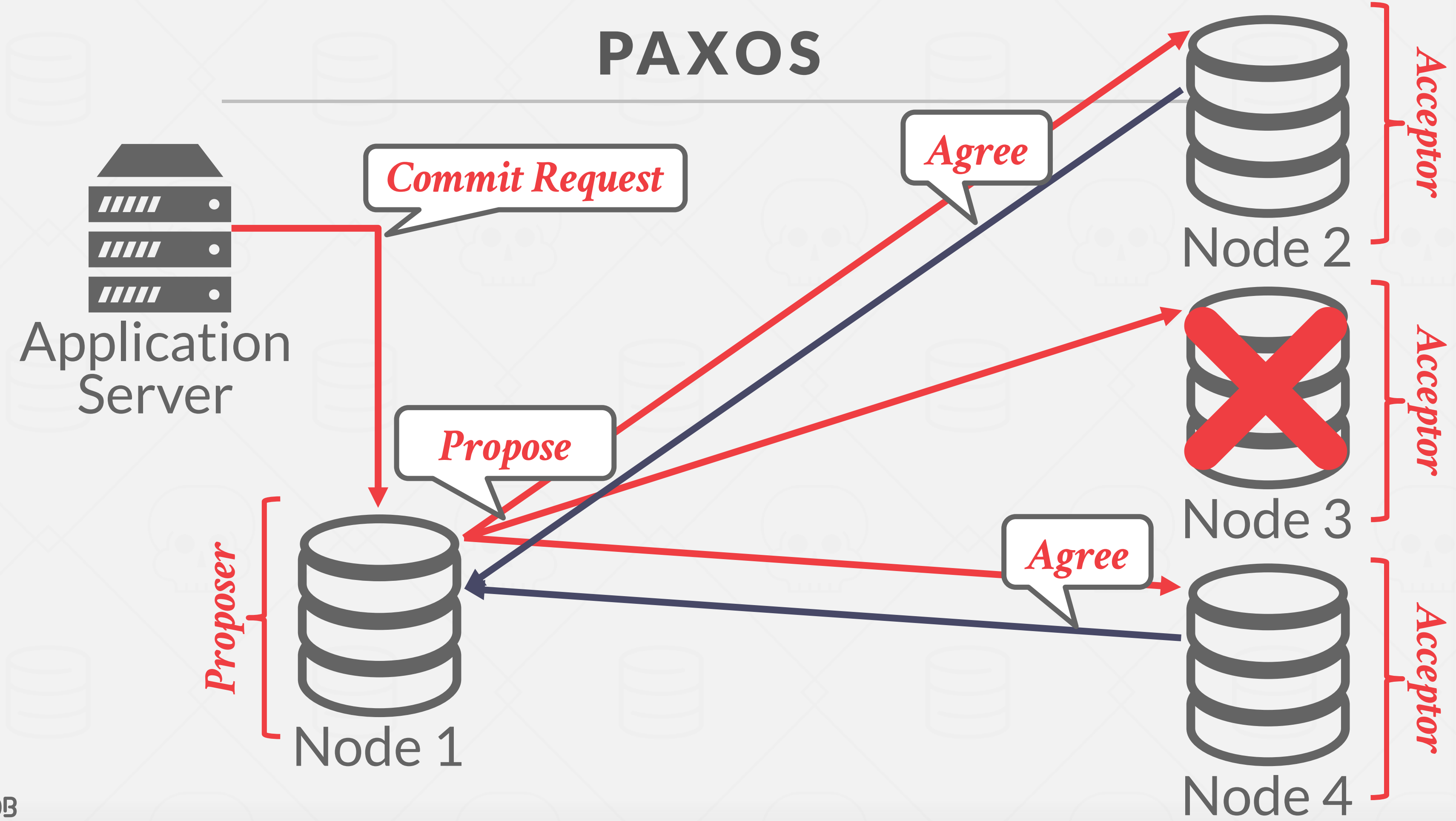

Commit Request
Application Server
Propose
Proposer
Node 1
Node 2
Acceptor
Node 3
Acceptor
Node 4
Acceptor

PAXOS

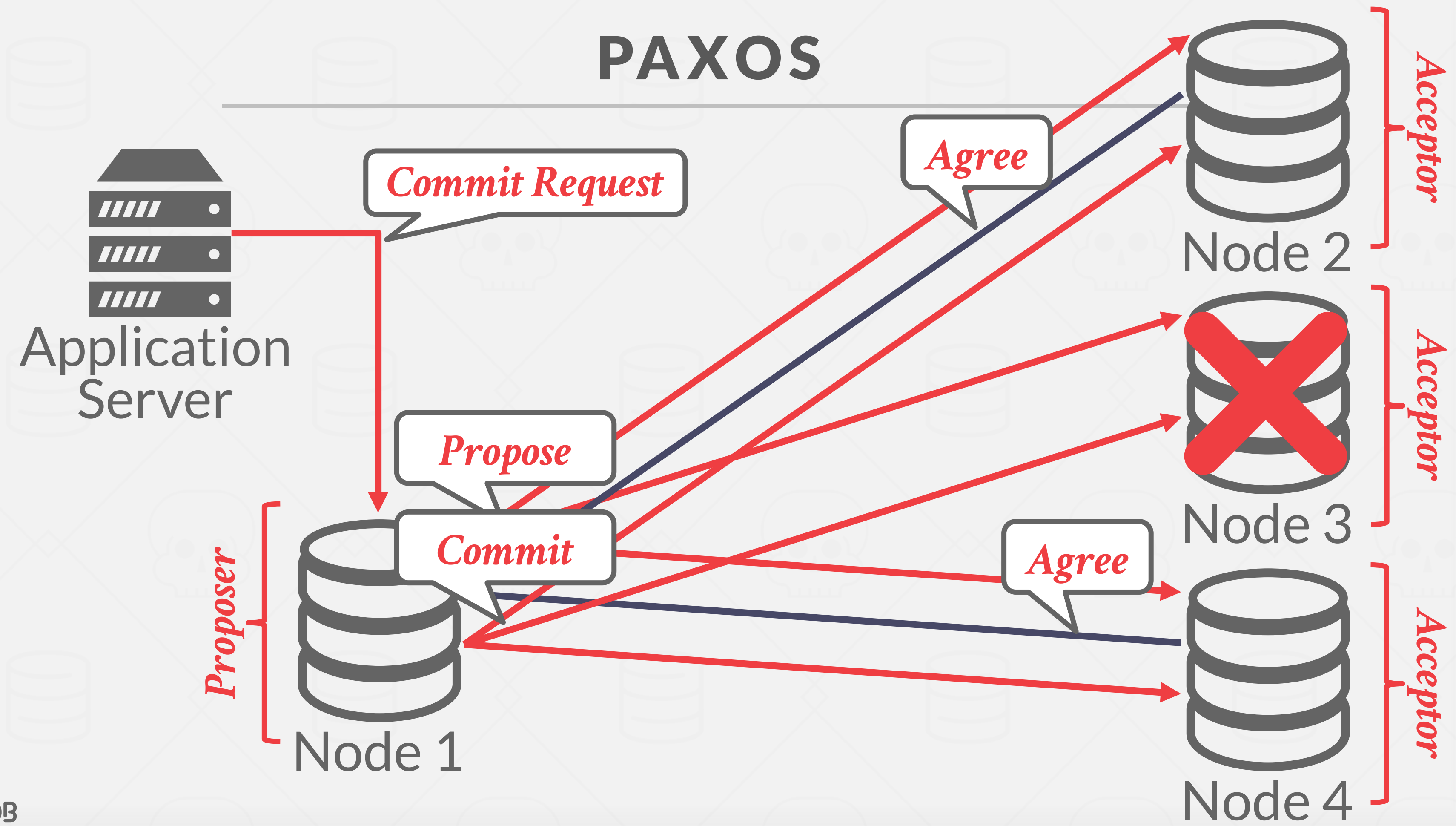


Commit Request
Application Server
Propose
Proposer
Node 1
Node 2
Acceptor
Node 3
Acceptor
Node 4
Acceptor

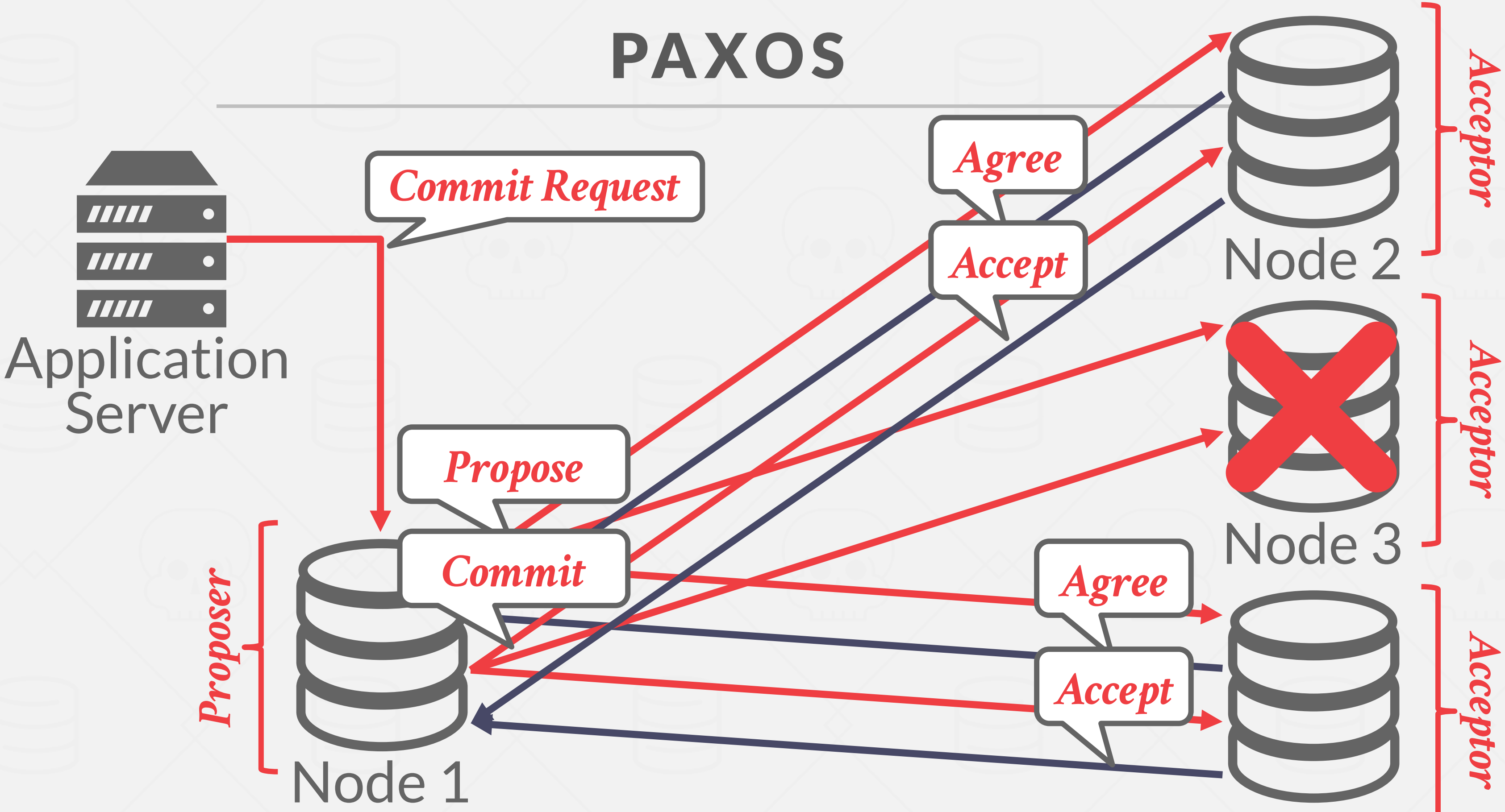
PAXOS


Application Server
Commit Request
Propose
Agree
Agree
Proposer
Node 1
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

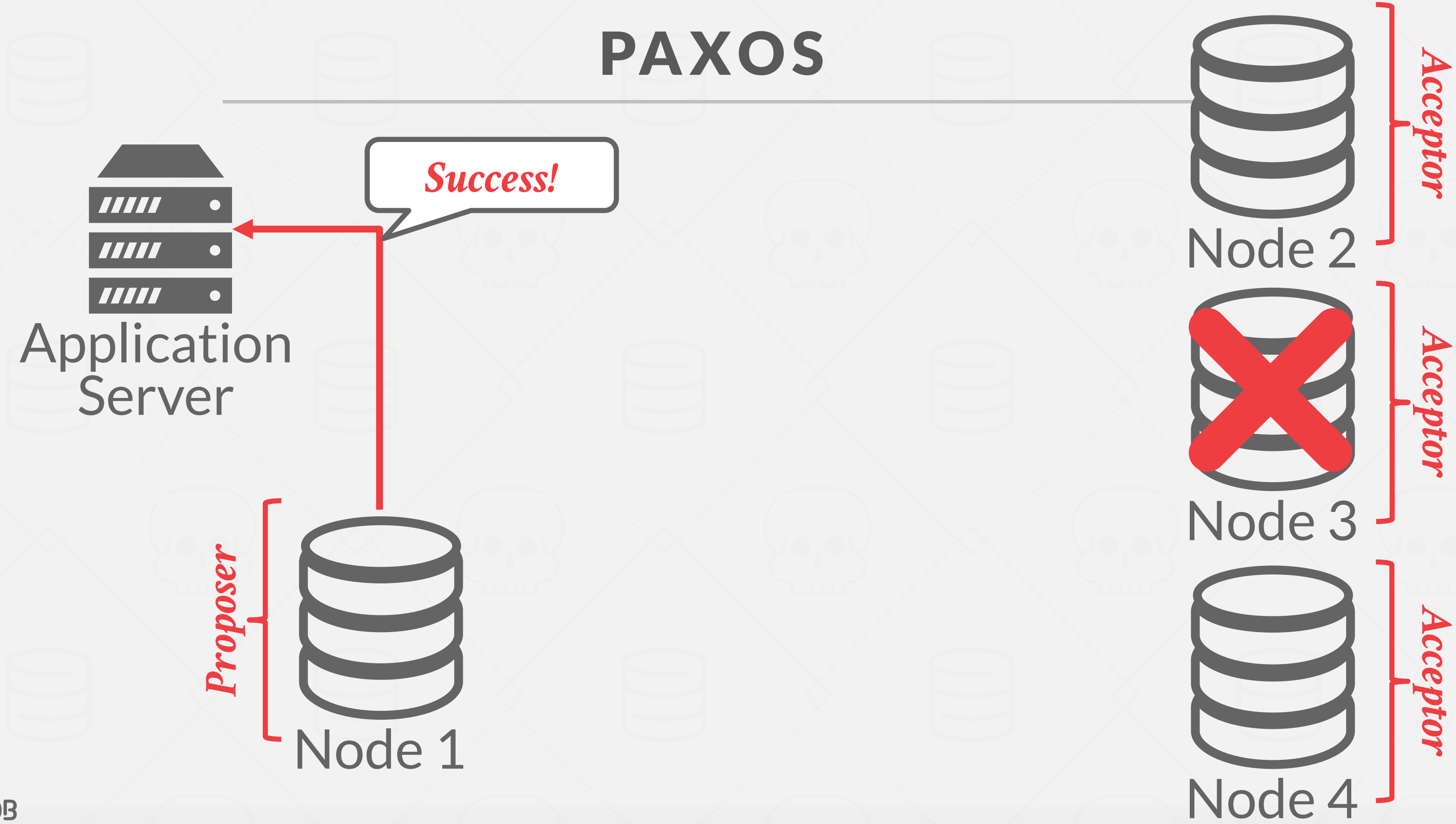
PAXOS


Application Server
Commit Request
Agree
Propose
Commit
Agree
Proposer
Node 1
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

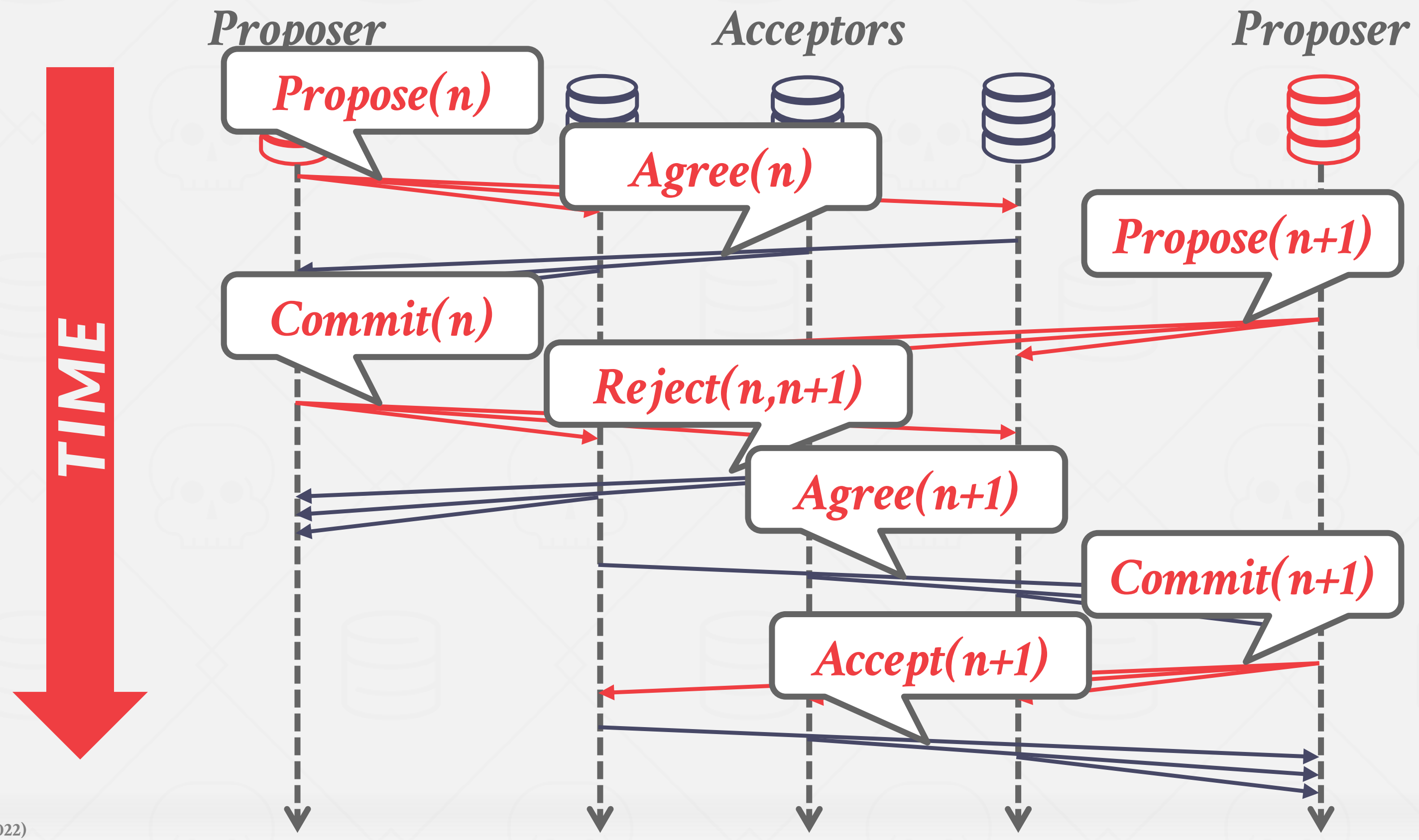
PAXOS


Application Server
Commit Request
Propose
Commit
Agree
Accept
Agree
Accept
Proposer
Node 1
Node 2
Node 3
Node 4
Acceptor
Acceptor
Acceptor

PAXOS


Success!
Application Server
Node 1
Proposer
Node 2
Acceptor
Node 3
Acceptor
Node 4
Acceptor

PAXOS


Proposer
Acceptors
Proposer
TIME
Propose(n)
Agree(n)
Propose(n+1)
Commit(n)
Reject(n,n+1)
Agree(n+1)
Commit(n+1)
Accept(n+1)

MULTI-PAXOS

If the system elects a single leader that oversees proposing changes for some period, then it can skip the **Propose** phase.

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

The system periodically renews the leader (known as a *lease*) using another Paxos round.

→ Nodes must exchange log entries during leader election to make sure that everyone is up-to-date.

2PC VS. PAXOS

Two-Phase Commit

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

Paxos

→ Non-blocking if 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
→ Update Method

REPLICA CONFIGURATIONS

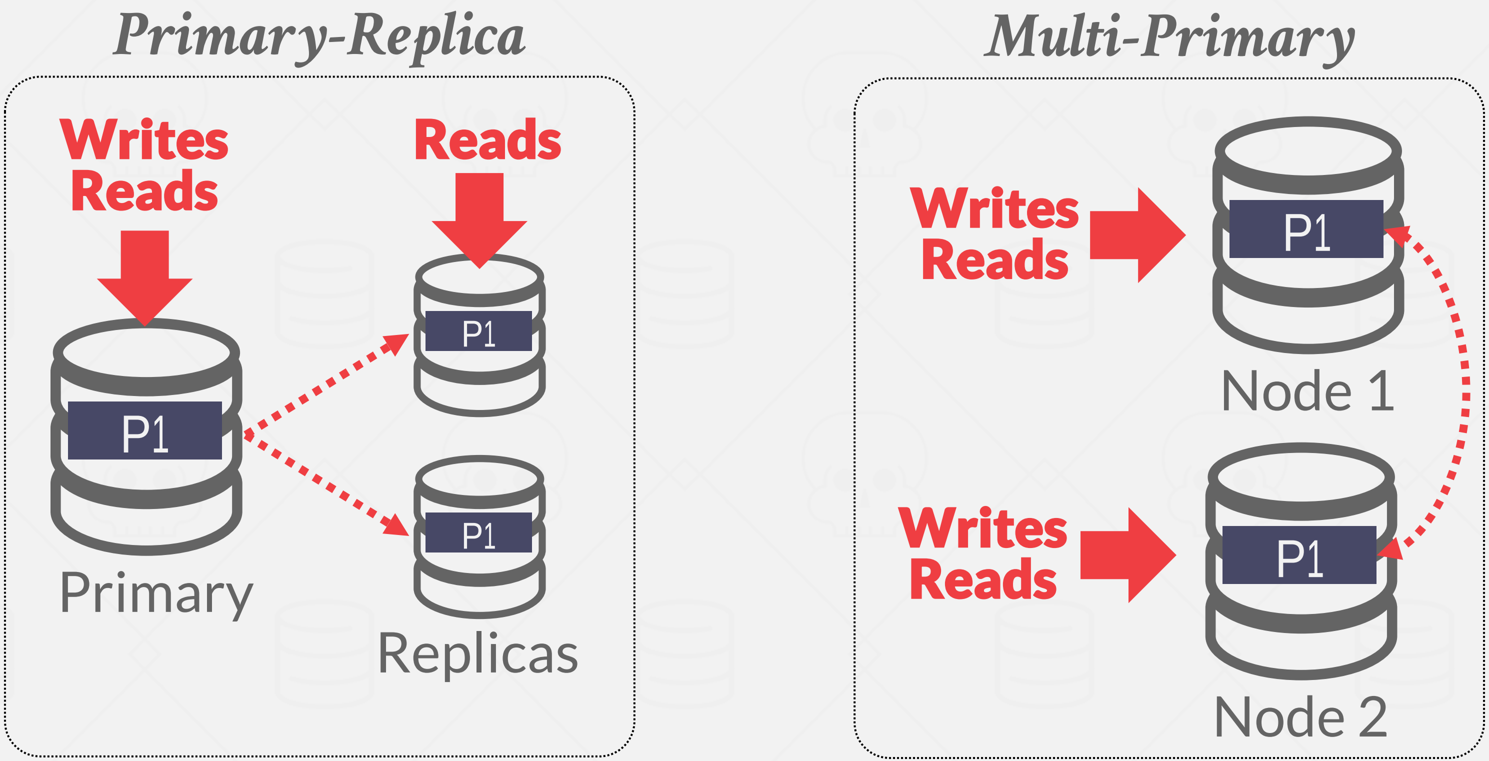
Approach #1: Primary-Replica

→ All updates go to a designated primary for each object.
→ The primary propagates updates to its replicas <u>without</u> an atomic commit protocol.
→ Read-only txns may be allowed to access replicas.
→ If the primary goes down, then hold an election to select a new primary.

Approach #2: Multi-Primary

→ Txns can update data objects at any replica.
→ Replicas <u>must</u> synchronize with each other using an atomic commit protocol.

REPLICA CONFIGURATIONS



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 always be available.

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

PROPAGATION SCHEME

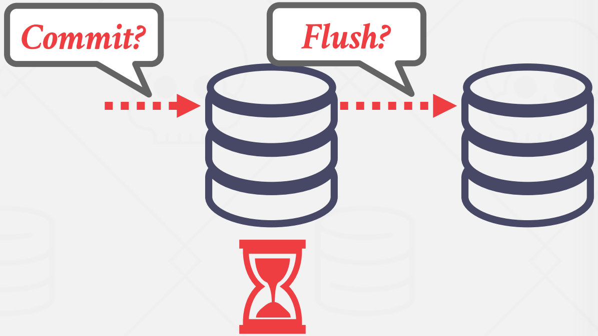
When a txn commits on a replicated database, the DBMS decides whether it must wait for that txn's changes to propagate to other nodes before it can send the acknowledgement to application.

Propagation levels:
→ Synchronous (*Strong Consistency*)
→ Asynchronous (*Eventual Consistency*)

PROPAGATION SCHEME

Approach #1: Synchronous

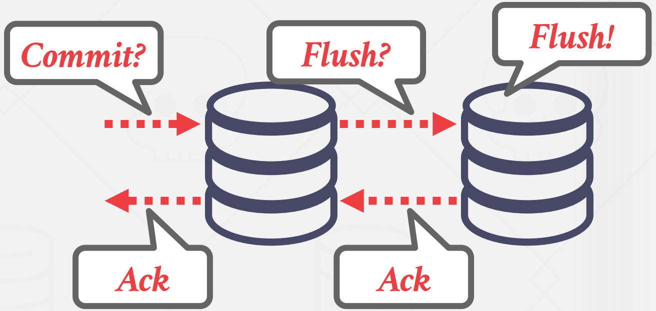
→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



PROPAGATION SCHEME

Approach #1: Synchronous

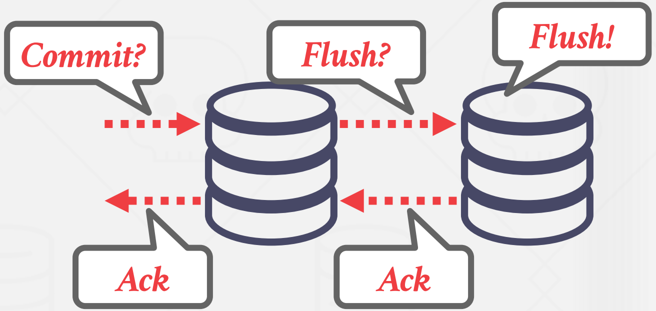
→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.



PROPAGATION SCHEME

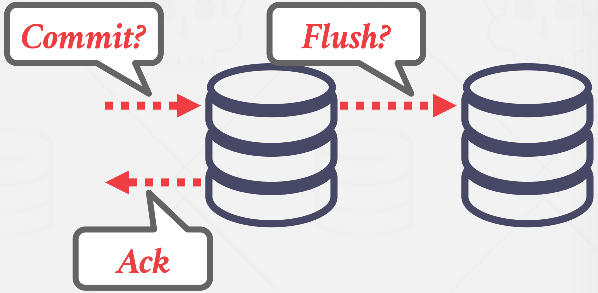
Approach #1: Synchronous

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



Approach #2: Asynchronous

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



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 records 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.
→ Can either do physical or logical replication.
→ Not the same as Primary-Replica vs. Multi-Primary

GOOGLE SPANNER

Google's geo-replicated DBMS (>2011)

Schematized, semi-relational data model.

Decentralized shared-disk architecture.

Log-structured on-disk storage.

Concurrency Control:

→ Strict 2PL + MVCC + Multi-Paxos + 2PC
→ **Externally consistent** global write-transactions with synchronous replication.
→ Lock-free read-only transactions.

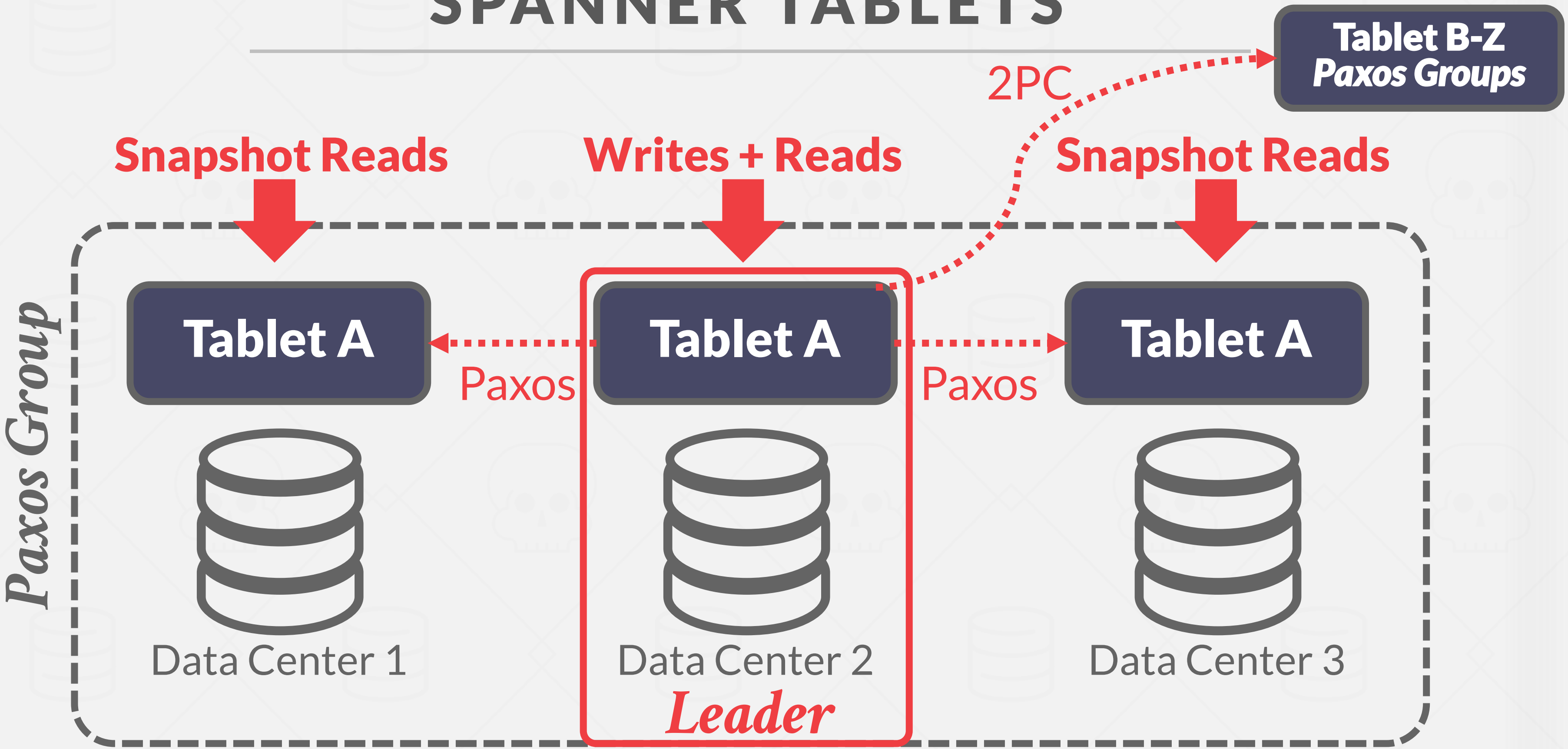
SPANNER: CONCURRENCY CONTROL

MVCC + Strict 2PL with Wound-Wait Deadlock Prevention

DBMS ensures ordering through globally unique timestamps generated from atomic clocks and GPS devices.

Database is broken up into tablets (partitions):
→ Use Paxos to elect leader in tablet group.
→ Use 2PC for txns that span tablets.

SPANNER TABLETS



SPANNER: TRANSACTION ORDERING

DBMS orders transactions based on physical "wall-clock" time.

→ This is necessary to guarantee strict serializability.
→ If T_1 finishes before T_2, then T_2 should see the result of T_1.

Each Paxos group decides in what order transactions should be committed according to the timestamps.

→ If T_1 commits at $time_1$ and T_2 starts at $time_2 > time_1$, then T_1's timestamp should be less than T_2's.

CAP THEOREM

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

→ Consistent
→ Always Available
→ Network Partition Tolerant

One flaw is that it ignores consistency vs. latency trade-offs.

→ See PACELC Theorem

Brewer

CAP THEOREM

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

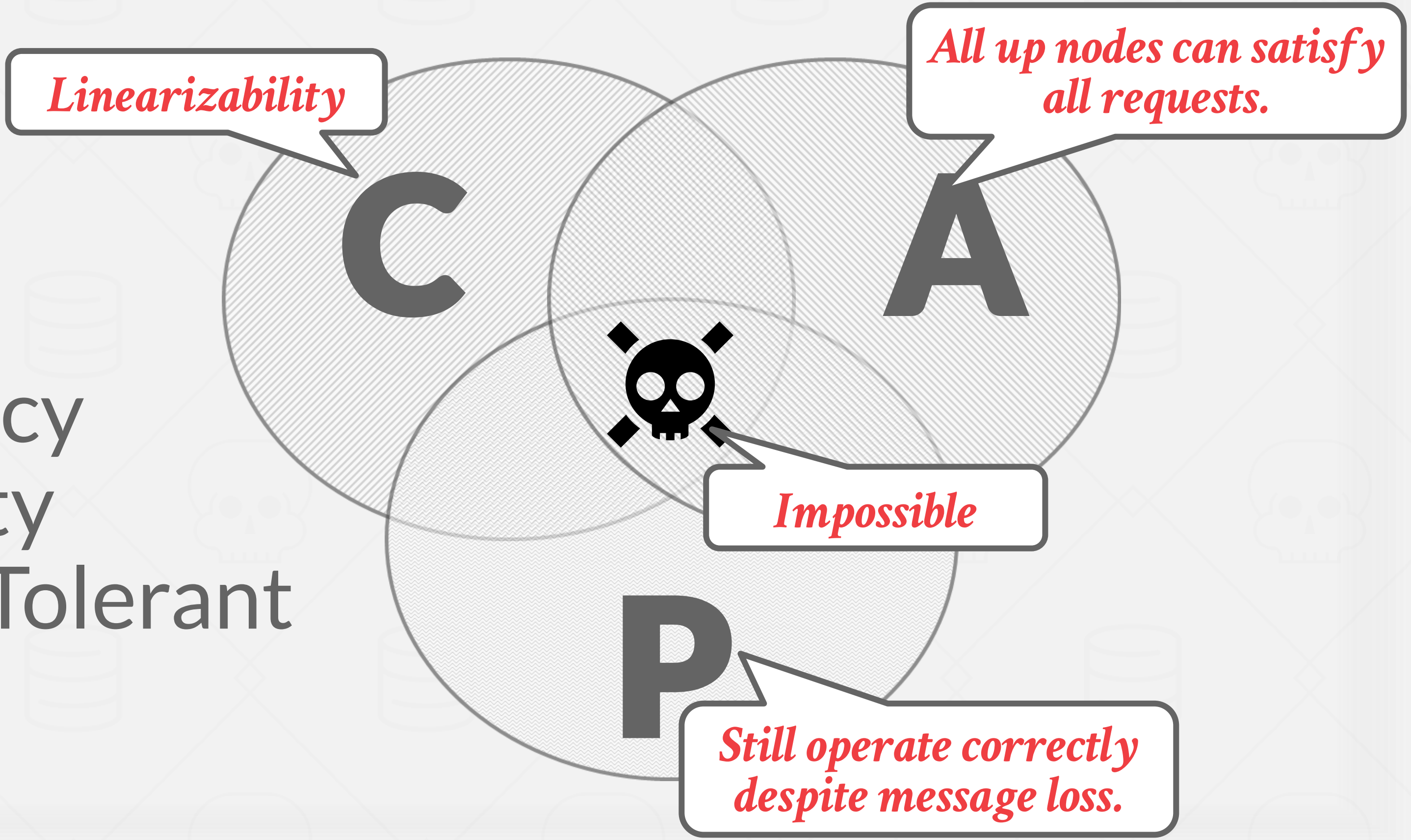
→ Consistent
→ Always Available
→ Network Partition Tolerant

One flaw is that it ignores consistency vs. latency trade-offs.

→ See PACELC Theorem

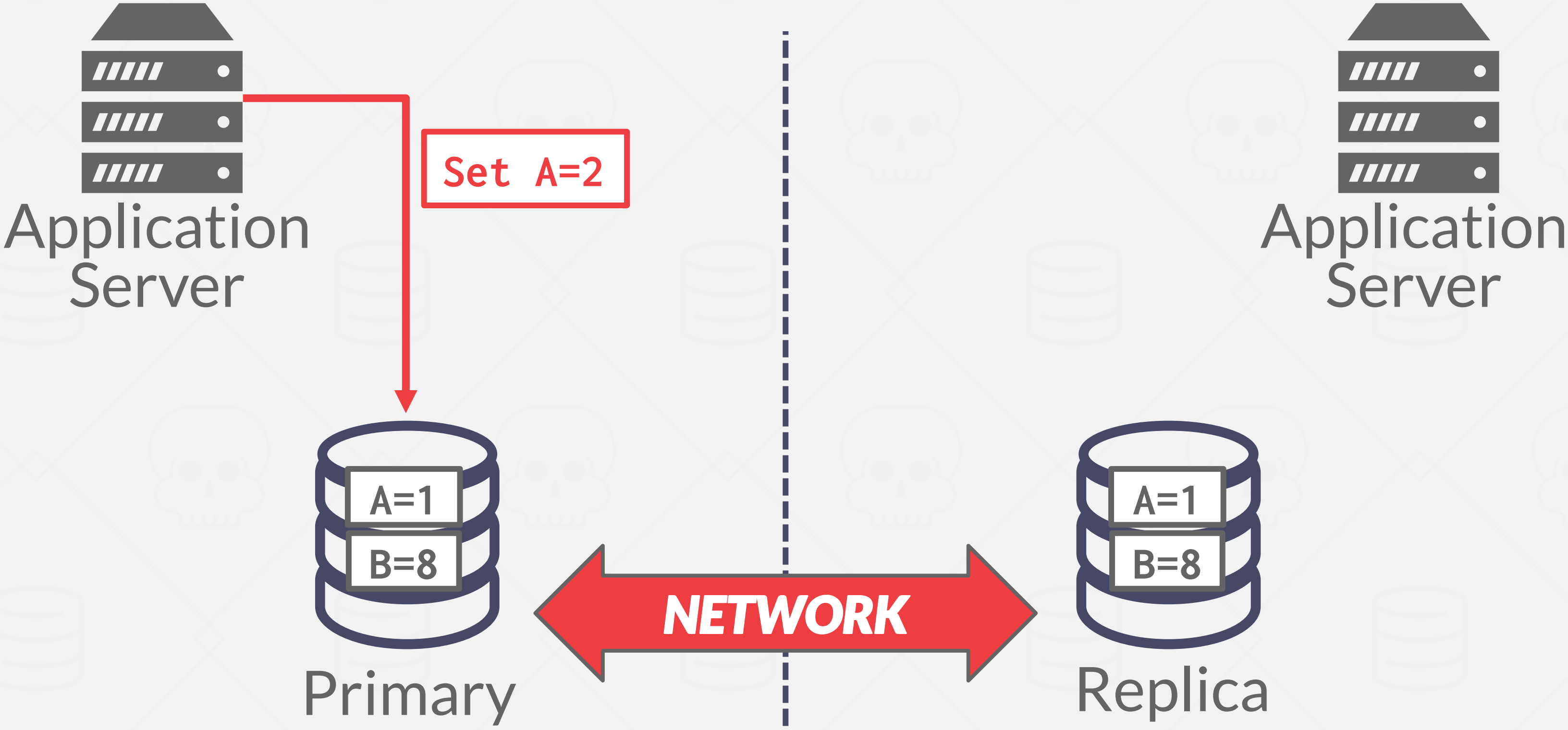
Brewer

CAP THEOREM

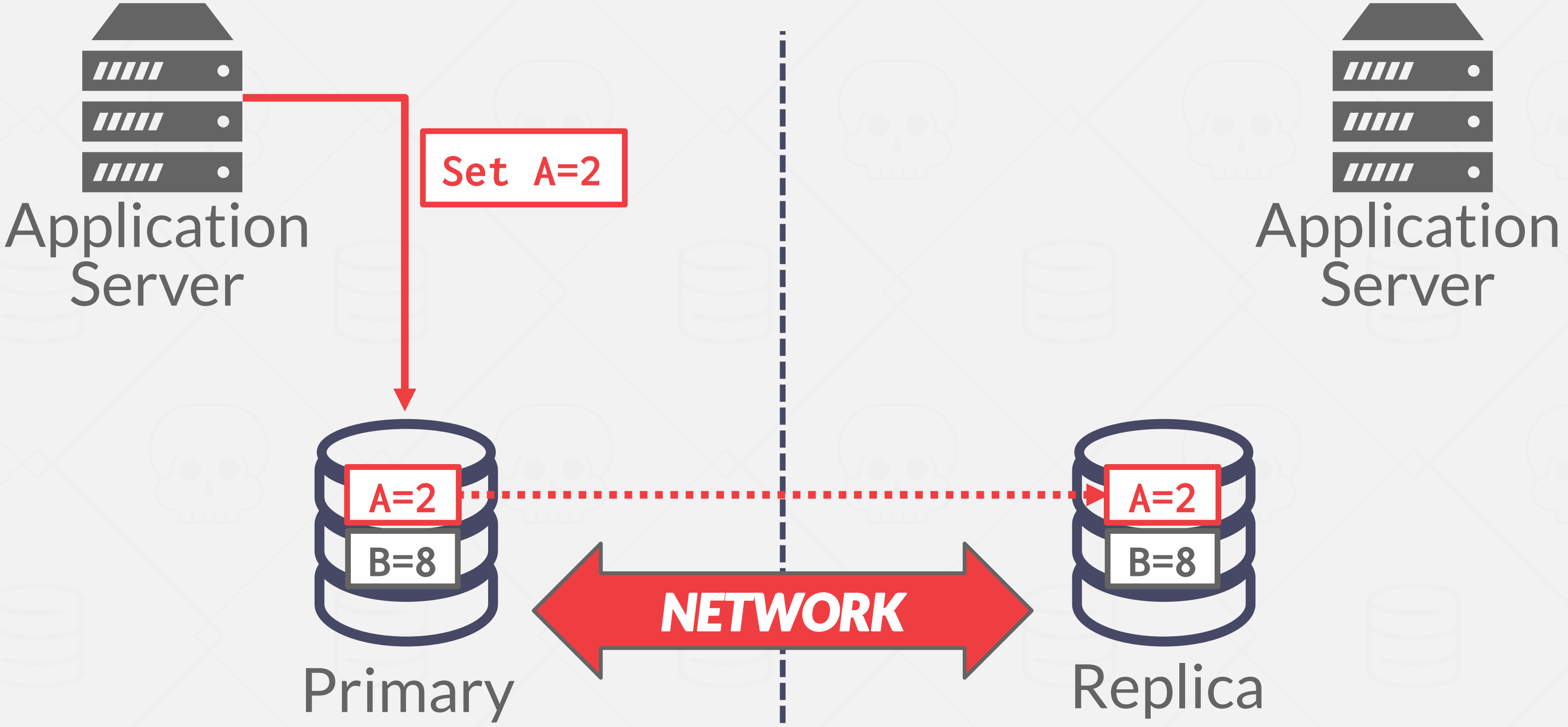


Consistency
Availability
Partition Tolerant

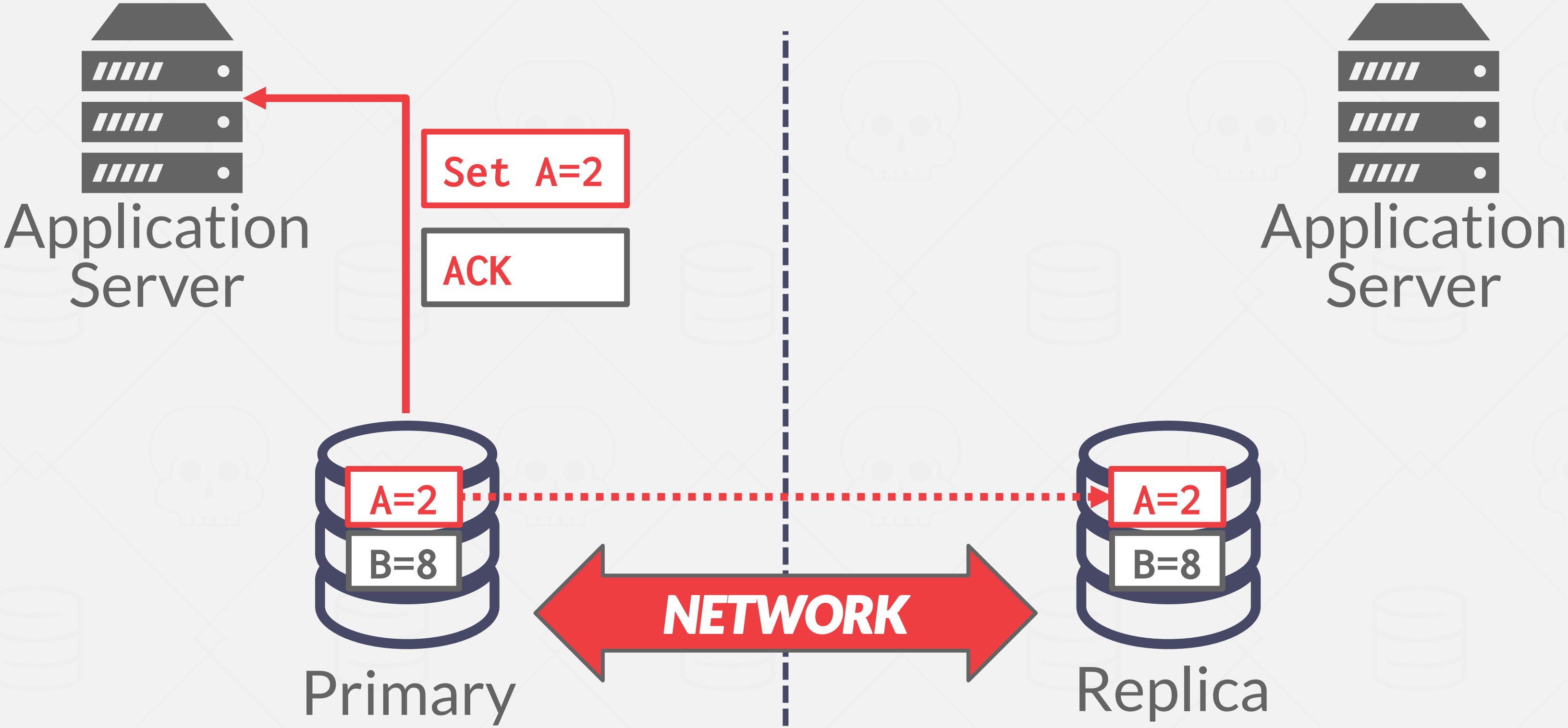
CAP – CONSISTENCY



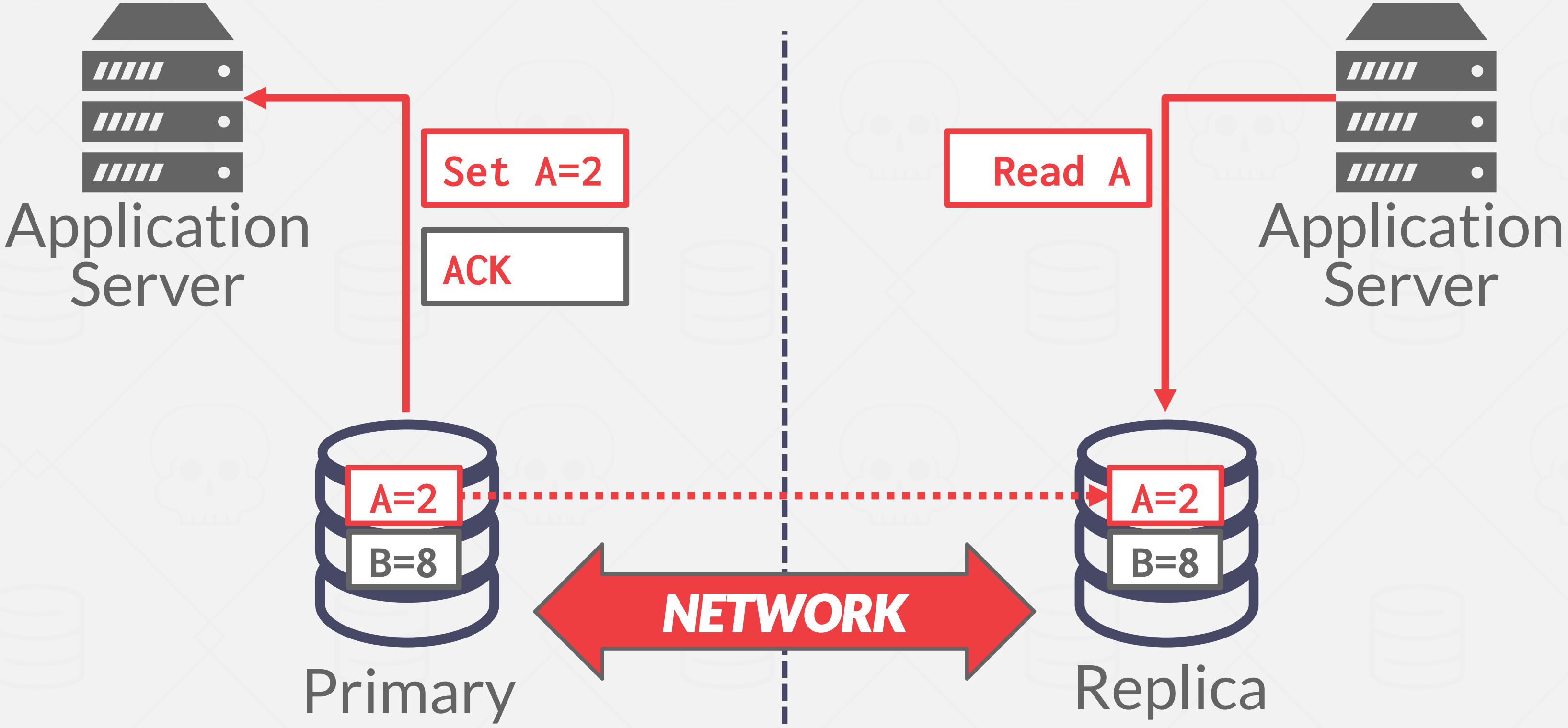
CAP – CONSISTENCY



CAP – CONSISTENCY

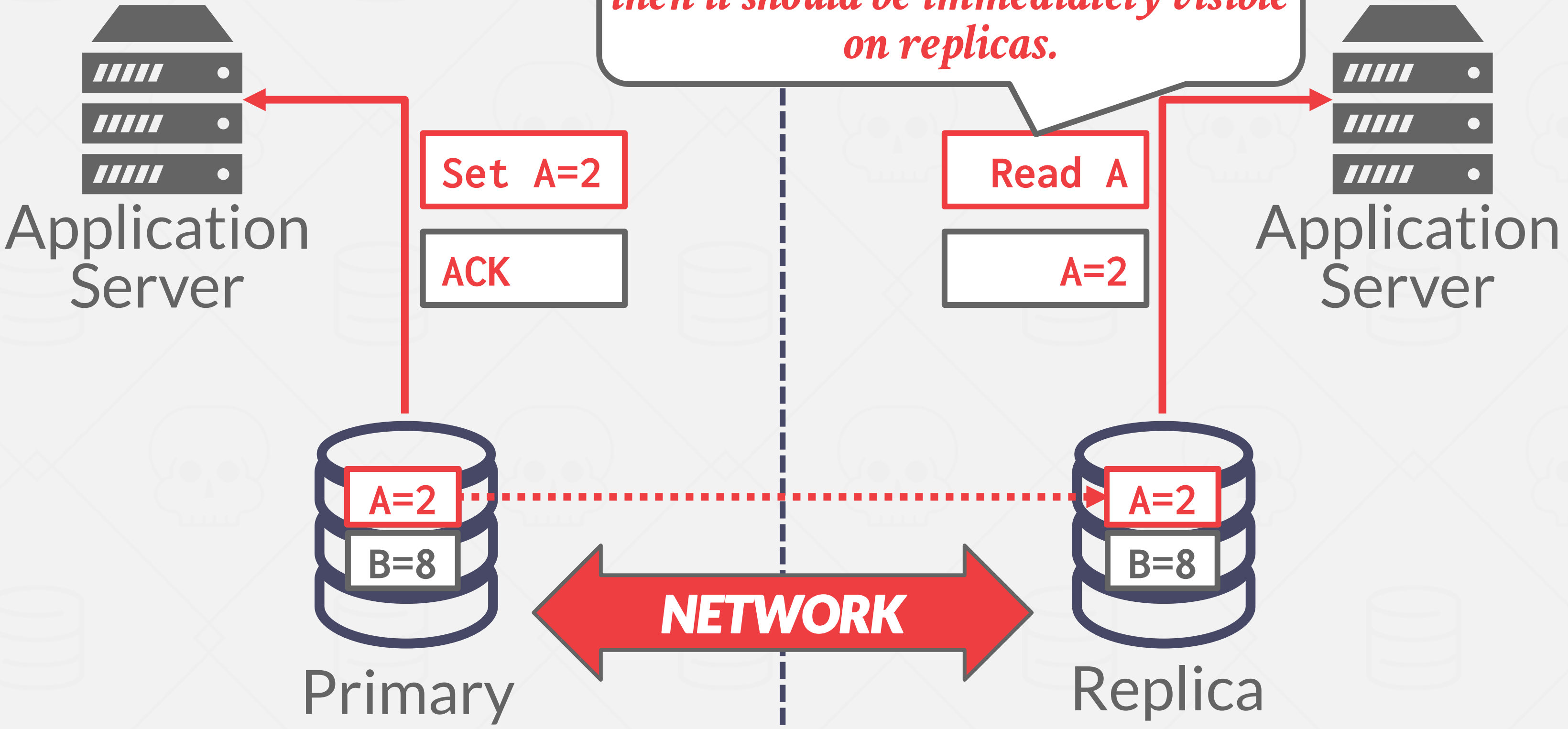


CAP – CONSISTENCY

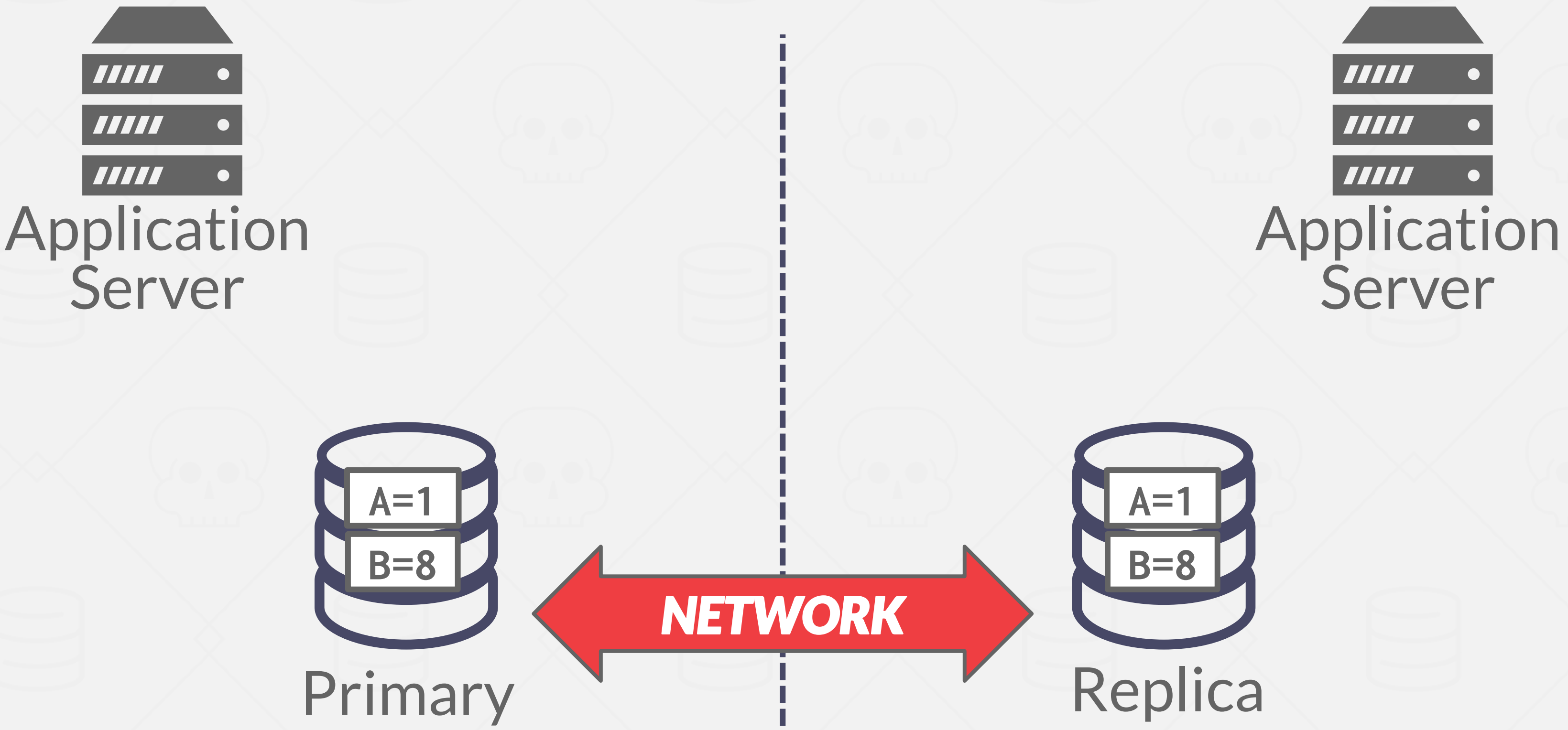


CAP –

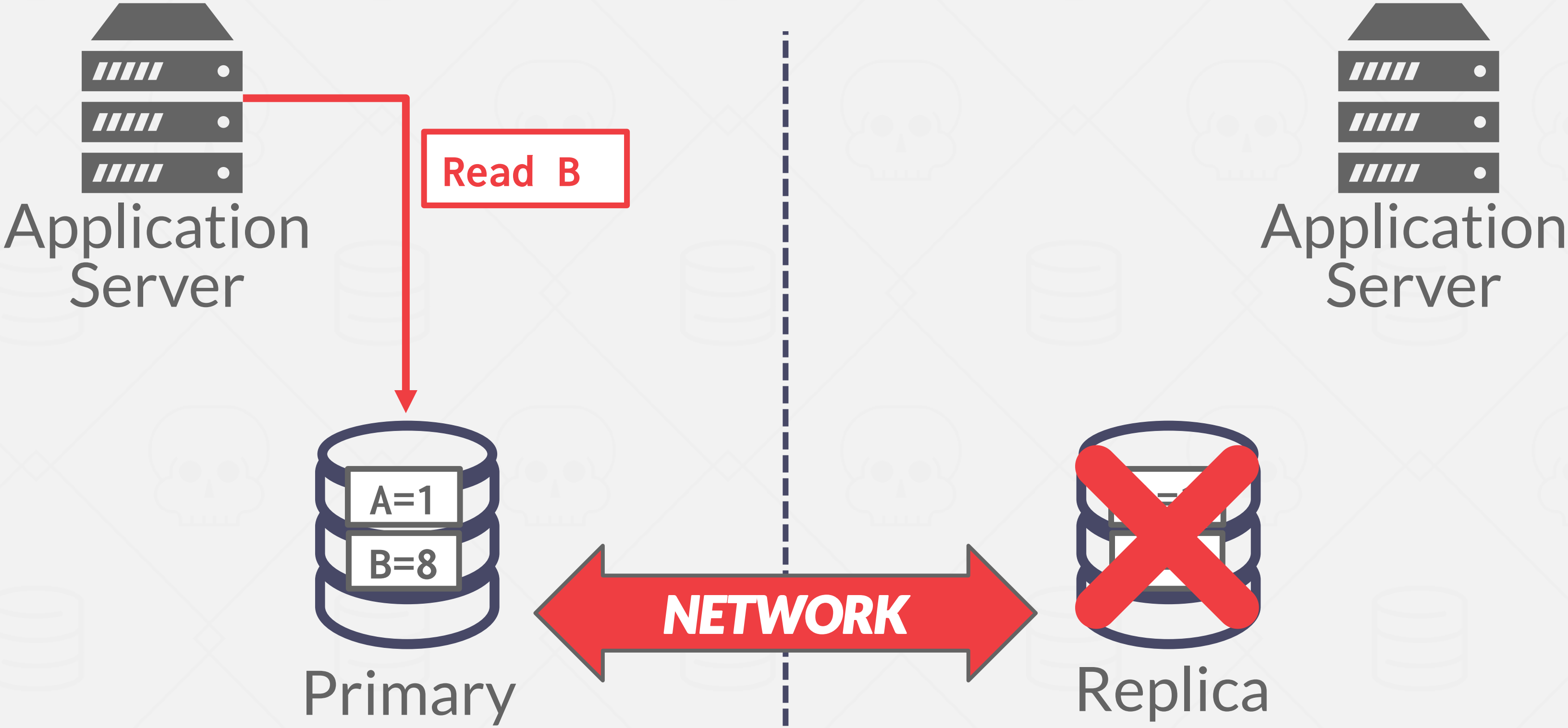
If Primary says the txn committed, then it should be immediately visible on replicas.



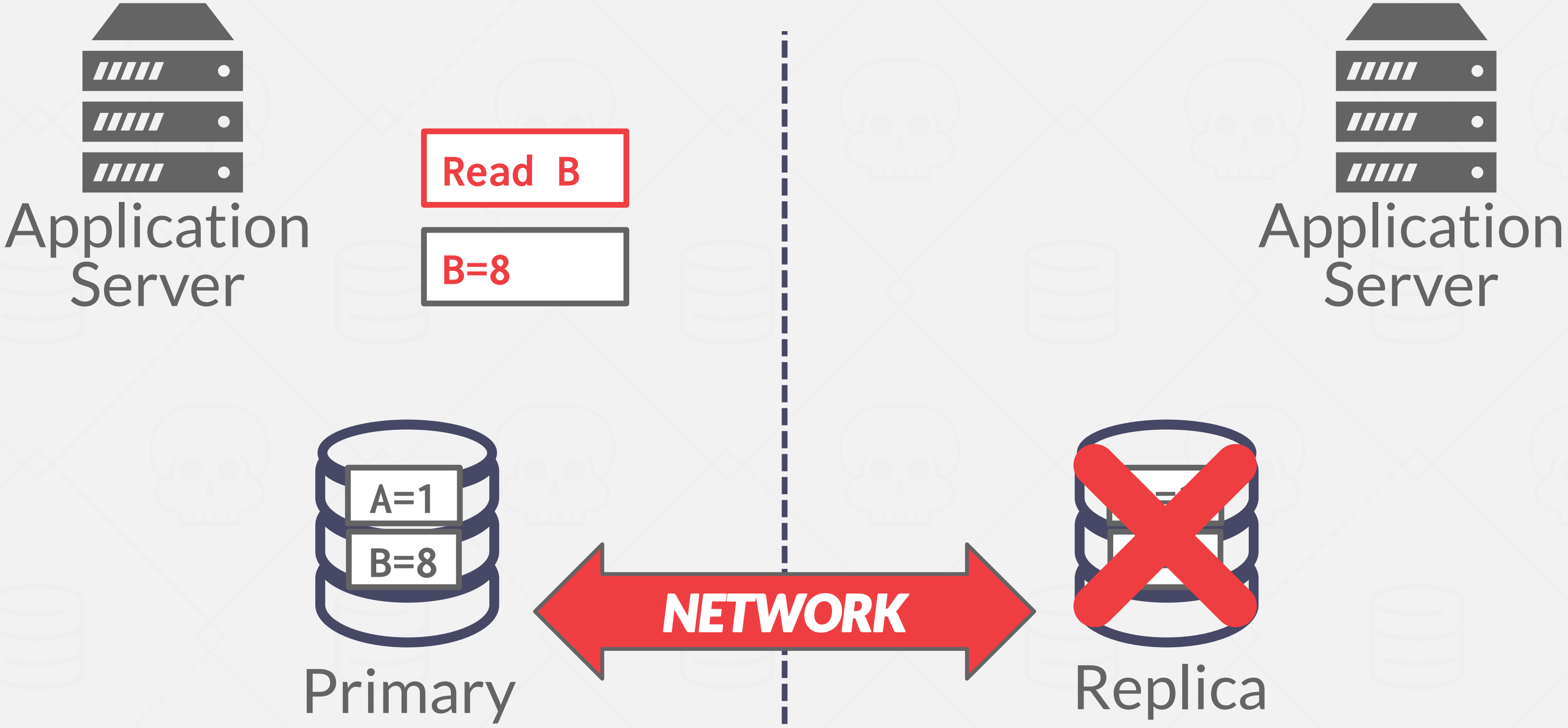
CAP – AVAILABILITY



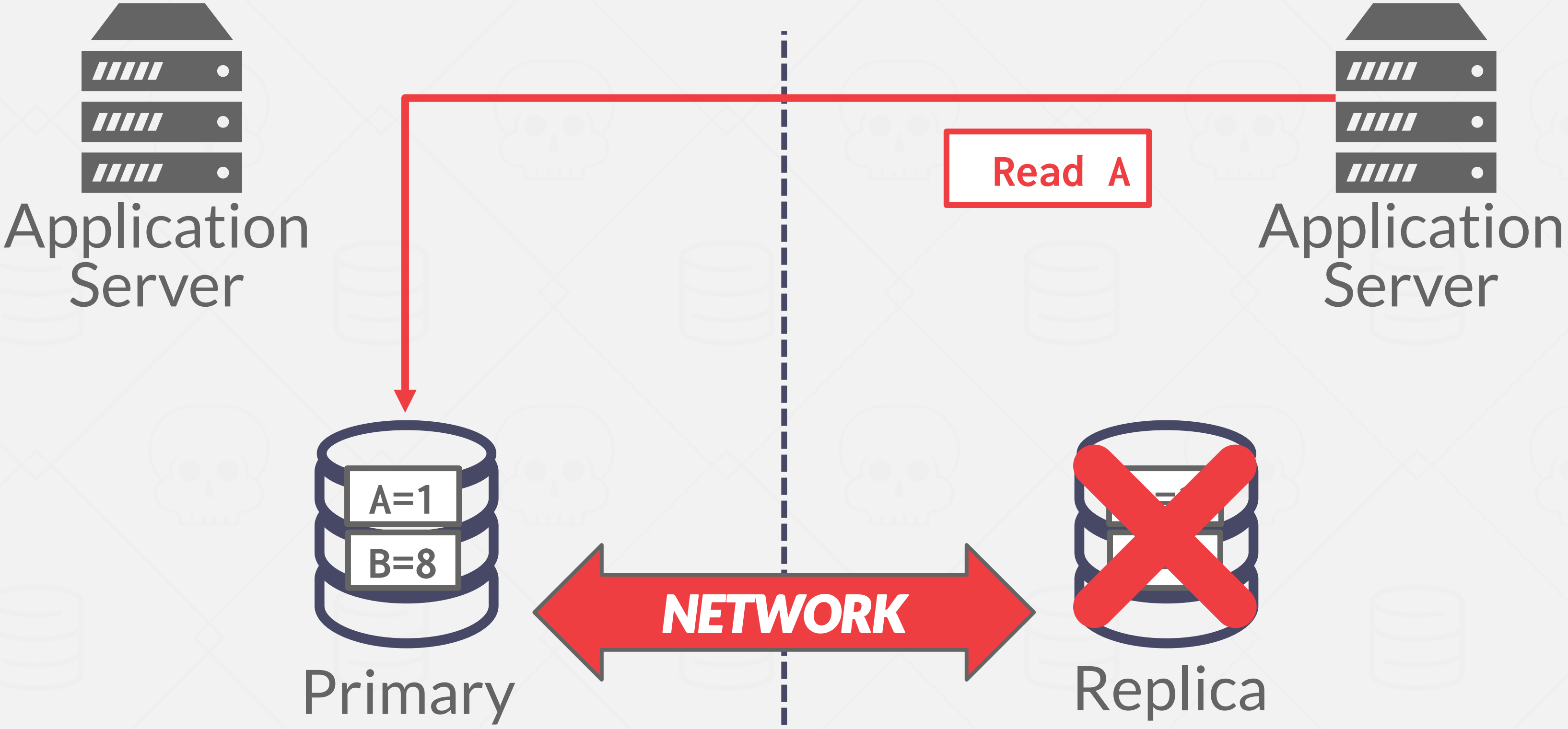
CAP – AVAILABILITY



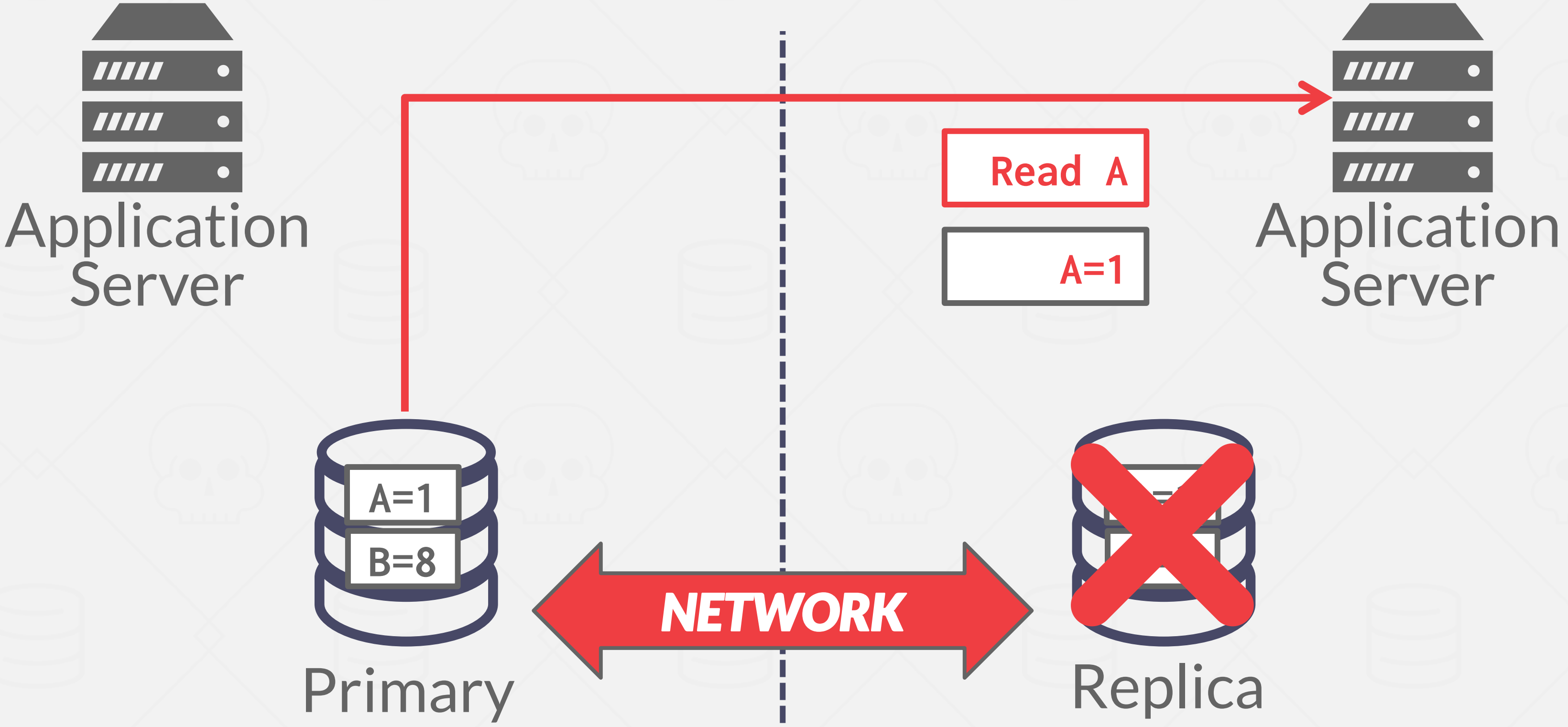
CAP – AVAILABILITY



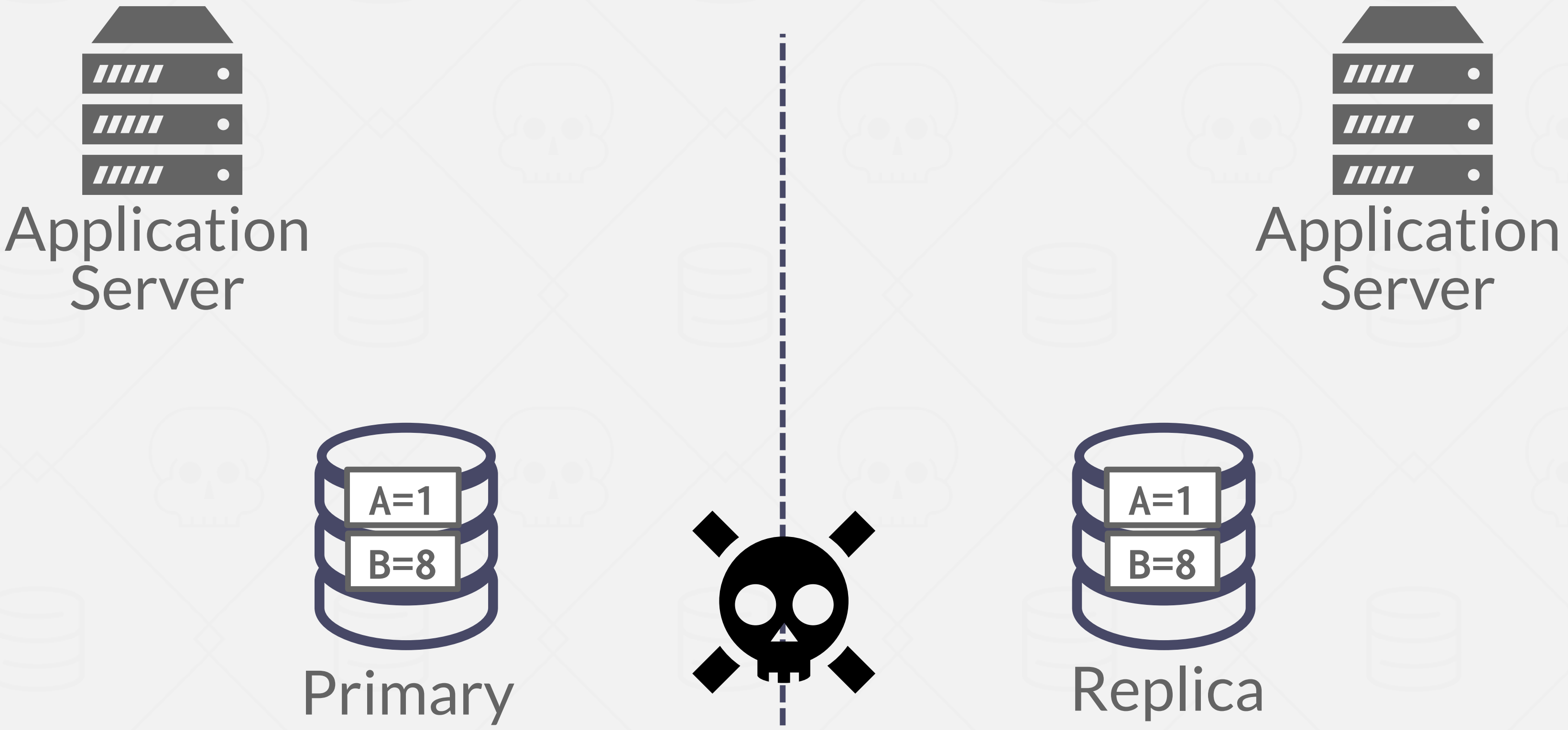
CAP – AVAILABILITY



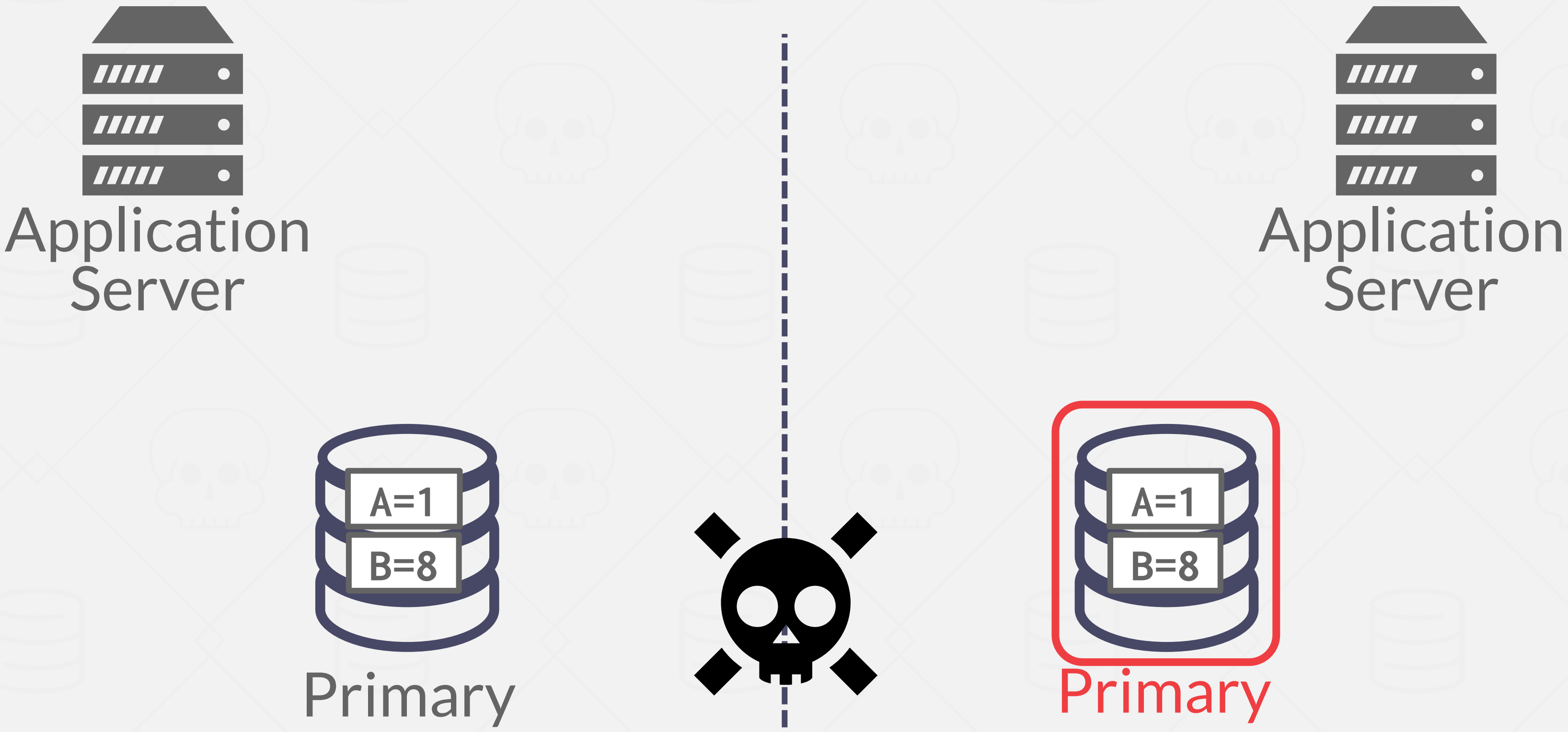
CAP – AVAILABILITY



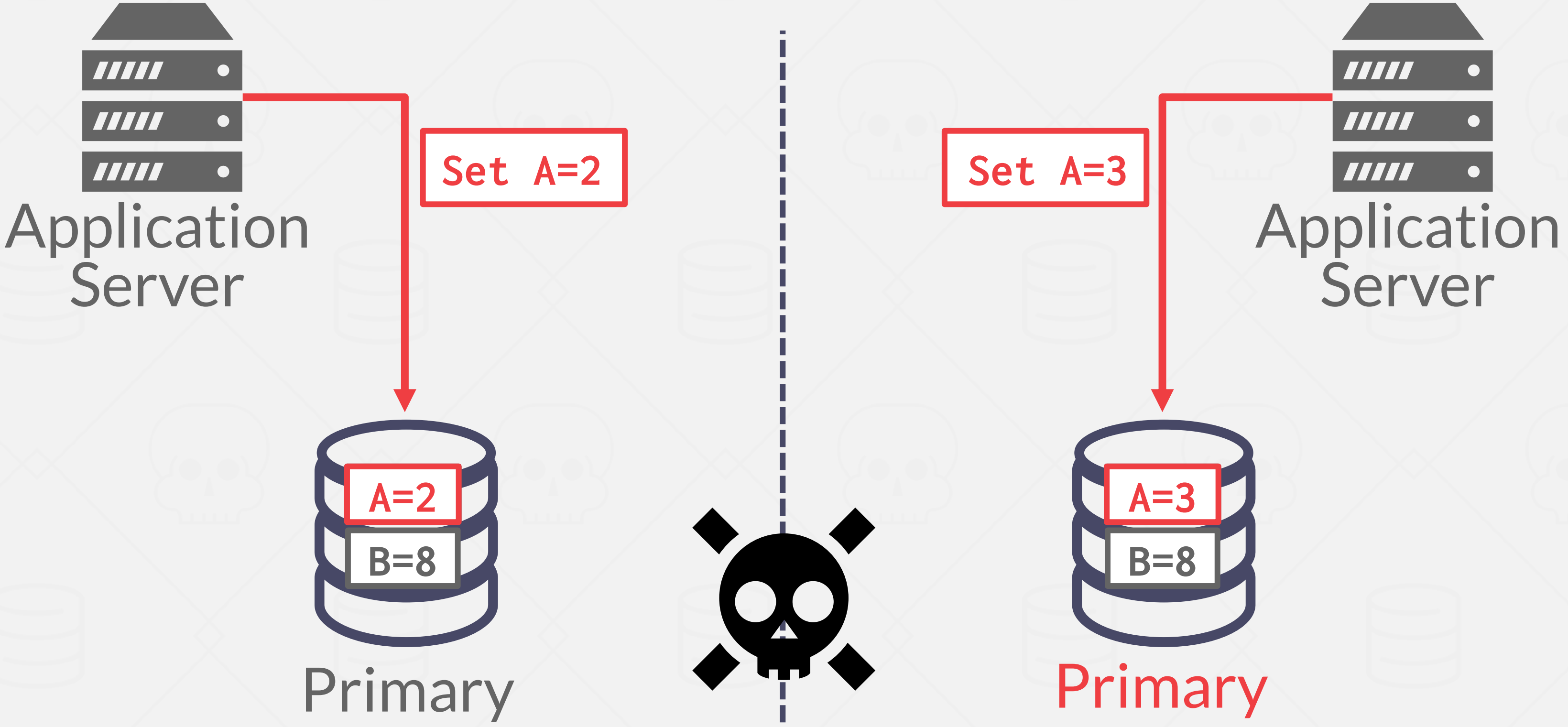
CAP – PARTITION TOLERANCE



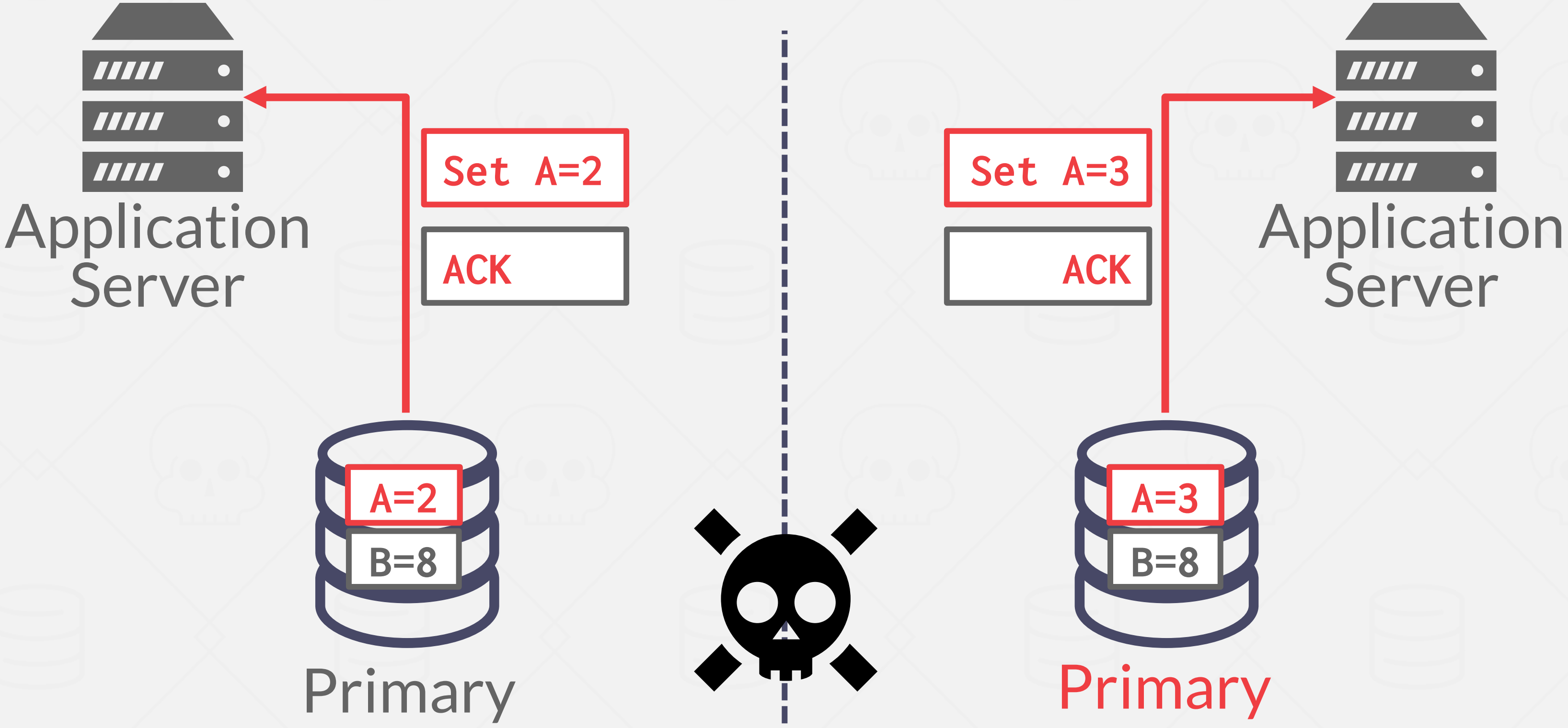
CAP – PARTITION TOLERANCE



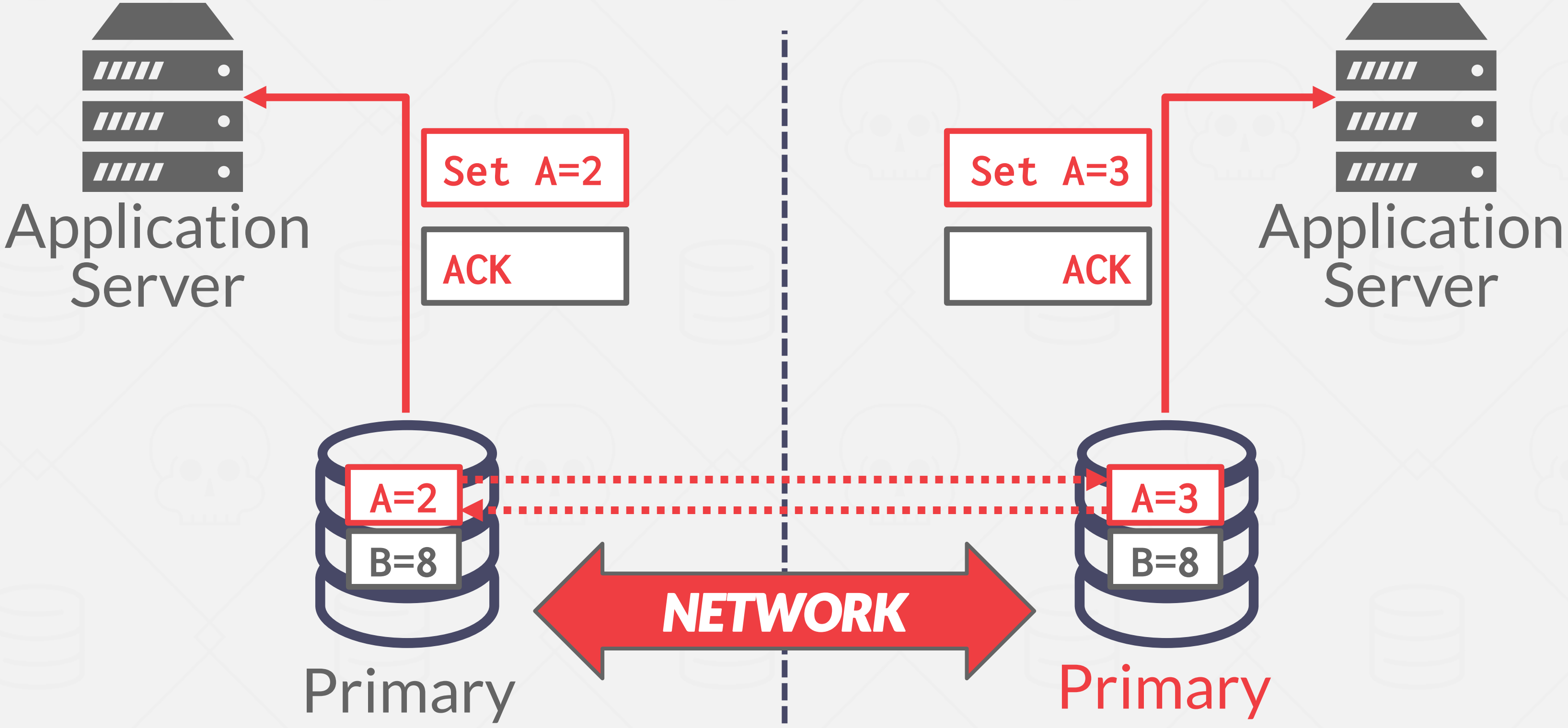
CAP – PARTITION TOLERANCE



CAP – PARTITION TOLERANCE



CAP – PARTITION TOLERANCE



CAP FOR OLTP DBMSs

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

Traditional/Distributed Relational DBMSs

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

NoSQL DBMSs

→ Provide mechanisms to resolve conflicts after nodes are reconnected.

CONCLUSION

Maintaining transactional consistency across multiple nodes is hard. Bad things will happen.

Blockchain databases assume that the nodes are adversarial. You must use different protocols to commit transactions. This is stupid.

More info (and humiliation):
→ Kyle Kingsbury's Jepsen Project

NEXT CLASS

Distributed OLAP Systems