Distributed Systems

Distributed Transactions and Commit Protocols

WEEK 14: Distributed Transactions, Two-Phase Commit, Atomicity, Consistency, Recovery Protocols.

Online Lecture Series - 14

Felix Edesa

Addis Ababa Science and Technology University

Course Outline: Distributed Transactions and Commit Protocols (Lecture 14)

Topics Overview

- **Distributed Transactions & ACID Principles** — Ensuring coordinated, reliable operations across multiple systems.
- **Commit Protocols: 2PC & 3PC** — Coordinator–participant models, phases.
- **Atomicity, Consistency & Recovery** — Checkpointing, logging.
- **Timeouts, Retries & Failure Handling** — Managing uncertain states and coordinator crashes.
- **Practical Applications** use microservices, and blockchain systems.

Distributed Transactions — Introduction

What Are Distributed Transactions?

- A single logical operation spanning multiple networked databases or microservices.
- Each part of the transaction executes on different machines or locations.
- Coordination is managed using a transaction coordinator.
- Guarantees consistency even when nodes are geographically distributed.
- Communication typically handled through commit protocols.

Why It Matters

- Ensures globally consistent outcomes.
- Prevents partial updates and data divergence.
- Common in e-commerce, finance, and IoT coordination.

ACID Principles in Distributed Transactions

Core Concepts

- **Atomicity** — Transactions complete entirely or not at all across all systems.
- **Consistency** — Ensures that distributed databases remain in valid states.
- **Isolation** — Parallel transactions behave as if executed sequentially.
- **Durability** — Once committed, results are saved even after failures.
- Distributed systems apply these using logging, coordination messages.

Example Scenario

- Booking a flight + hotel: both must confirm or cancel together.
- Realized using a distributed transaction manager coordinating multiple services.

Atomicity — All or Nothing

Concept and Implementation

- Atomicity ensures no intermediate or partial updates exist.
- Uses a "commit or rollback" approach across all nodes.
- Implemented through the **Two-Phase Commit (2PC)** protocol:
  - Phase 1: Coordinator asks participants to prepare.
  - Phase 2: Coordinator sends commit or abort decision.

Practical Example

- Online order system:
  - Deduct payment.
  - Update stock.
  - Generate receipt.
- Failure in one $\rightarrow$ rollback all.

Consistency — Maintaining Valid States

Definition and Mechanism

- Ensures distributed nodes transition from one valid state to another.
- Prevents conflicting or invalid updates due to concurrency.
- Achieved via:
  - Constraints and invariants enforced globally.
  - Replication protocols (quorums, Paxos-based consensus).
  - Validation checks before commit.

Example

- Bank transfer must maintain total balance constant.
- Violations can cause duplicated or missing funds.

Isolation — Handling Concurrency

Mechanism

- Controls visibility of transaction operations to others.
- Isolation levels:
  - Read Uncommitted
  - Read Committed
  - Repeatable Read
  - Serializable
- Tools used:
  - Locking and versioning (MVCC).
  - Timestamp ordering and validation.
- Prevents dirty reads, lost updates.

Example

- Two users adding to same cart — system must serialize updates.
- Database ensures updates don't conflict.

Durability — Surviving Failures

Concept

- Once a transaction commits, results must persist permanently.
- Managed through:
  - Write-ahead logging (WAL)
  - Checkpoints
  - Distributed persistent storage
- Recovery process replays logs to rebuild last consistent state.
- Essential for fault-tolerant distributed databases.

Example

- Payment confirmation logged before user notified.
- After crash, log replay restores final committed state.

ACID in Microservices Environments

Challenges and Adaptations

- Microservices own independent databases — global ACID is hard.
- Services communicate asynchronously via APIs or message queues.
- Instead of strict ACID:
  - Use **Saga Pattern** for distributed coordination.
  - Implement compensating transactions.
- 2PC may cause blocking; Saga is more scalable.

Example

- E-commerce checkout: order, payment, and shipping as separate services.
- Compensation cancels payment if shipment fails.

Real-World Scenario: Distributed E-Commerce Checkout

Workflow Across Services

- **Inventory Service:** reserves product.
- **Payment Service:** processes charge.
- **Order Service:** confirms order and records.
- **Notification Service:** emails receipt.
- Each component participates in one distributed transaction cycle.
- Any component failure triggers rollback or compensation.

Lesson

- Reliability requires orchestration between all services.
- Atomicity ensures no "orphan" orders or double charges.

Case Study — Distributed Banking Transactions

Scenario and Process

- Funds transfer between Bank A and Bank B.
- Coordinator initiates "Prepare to Commit" message to both.
- Each bank verifies local operation feasibility.
- If both ready $\rightarrow$ commit. Otherwise, rollback.
- Achieved via 2PC or 3PC for higher reliability.
- Recovery logs ensure no duplicate debit or credit occurs.

Key Benefit

- Guarantees global consistency of financial records.
- Prevents one-sided transaction outcomes.

Challenges in Distributed ACID Enforcement

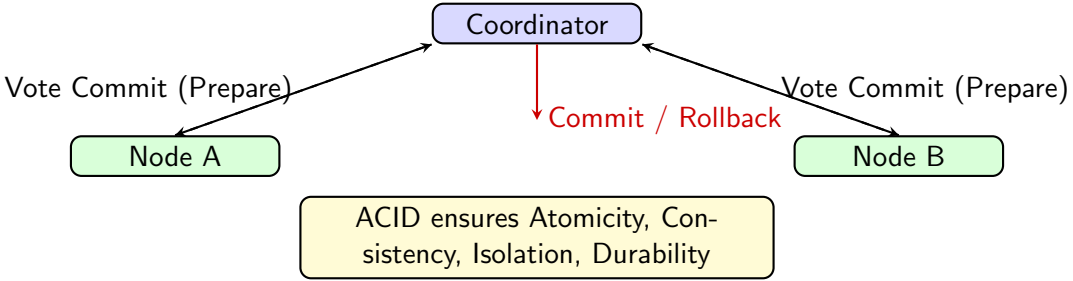
Common Challenges

- **Network Partition:** Delays or message loss.
- **Coordinator Failure:** Participants stuck in prepare state.
- **Performance Overhead:** Multiple message rounds reduce throughput.
- **Clock Synchronization:** Makes ordering and isolation complex.
- **Trade-Off:** Strong consistency vs. high availability (CAP theorem).

Mitigation

- Fault-tolerant commit protocols (3PC, Paxos Commit).
- Checkpoints and timeouts for recovery.
- Using consensus models (Raft, Zab).

Distributed Transactions ACID Principles: Summary



Summary — Distributed Transactions and ACID

Key Takeaways

- Distributed transactions preserve global consistency across systems.
- ACID ensures predictable, reliable operation under failure.
- Commit and recovery protocols handle distributed uncertainty.
- Used in real-world domains: banking, logistics, and microservices orchestration.

Commit Protocols: Introduction

What are Commit Protocols

- Ensure all nodes in a distributed transaction **agree on commit or rollback**.
- Handle failures and uncertain states in distributed systems.
- Maintain **atomicity and consistency** across multiple participants.
- Key protocols: **Two-Phase Commit (2PC)** and **Three-Phase Commit (3PC)**.

Why it matters

- Prevents partial updates across distributed databases.
- Ensures reliable operations in banking, logistics, and microservices.

Two-Phase Commit (2PC): Overview

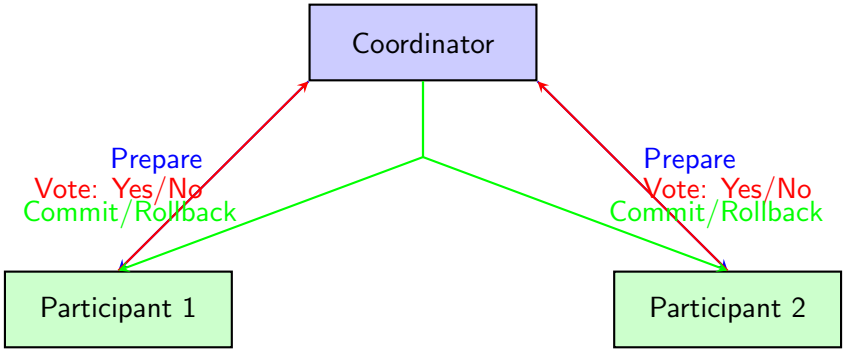
Key Concepts

- **Coordinator** sends a "Prepare" request to participants.
- Participants vote: **Yes (ready to commit)** or **No (abort)**.
- Coordinator decides **Commit** if all Yes; otherwise **Rollback**.
- Guarantees **atomicity** even if failures occur, with blocking potential.

Practical Example

- Transferring money between bank branches.
- Updating inventory across multiple warehouses.

2PC Coordinator-Participant Diagram



2PC Failure Handling

Coordinator Crash Example

- Coordinator crashes after participants vote Yes.
- Participants are **blocked until coordinator recovers**.
- Logs are used to **determine commit/rollback upon recovery**.
- Highlights **blocking problem** in 2PC.

Practical Example

- Bank transaction stuck because central server failed.
- Must ensure no partial money transfer occurs.

Three-Phase Commit (3PC): Introduction

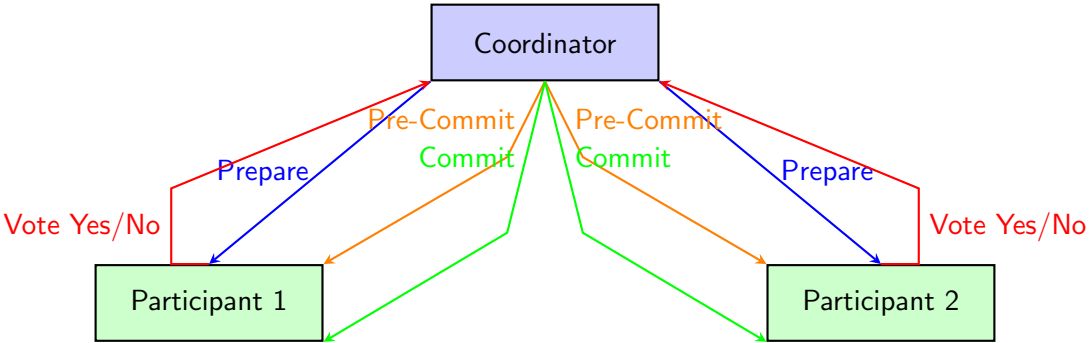
Key Features

- Introduces an extra **pre-commit phase** to avoid blocking.
- Coordinator sends **Prepare**, participants respond, then **Pre-Commit** is sent.
- Guarantees **non-blocking commit** under coordinator failures.
- Improves fault-tolerance and handles **network partitions**.

Example

- Microservices transaction across distributed services.
- Logistics updates across regional warehouses without blocking operations.

3PC Coordinator-Participant Diagram



3PC Failure Handling

Non-Blocking Example

- Coordinator crashes during pre-commit phase.
- Participants can **proceed to commit or rollback safely** based on timeout and pre-commit logs.
- Prevents indefinite blocking seen in 2PC.

Example

- Payment microservice still completes transaction even if central coordinator fails.

2PC vs 3PC Comparison

Key Differences

- **2PC** may block participants if coordinator fails.
- **3PC** adds pre-commit phase to avoid blocking.
- **3PC** handles network partitions more gracefully.
- Both maintain **atomicity** and **consistency**.

Use Cases

- Banking systems: 2PC for simple branch coordination.
- Microservices: 3PC for resilient, distributed workflows.

Distributed Transaction in Banking

Scenario

- Transferring money across multiple bank branches.
- Steps: debit from source, credit to destination, update logs.
- Commit protocol ensures either **all steps succeed** or **none**.
- Guarantees **no partial transaction**, even if failures occur.

Lesson

- Demonstrates criticality of commit protocols for **global consistency**.

Key Takeaways: Commit Protocols

Summary

- Distributed transactions preserve **global consistency**.
- **ACID principles** ensure predictable, reliable operations.
- 2PC and 3PC handle failures and uncertain states differently.
- Widely used in **banking, logistics, and microservices orchestration**.

Reminder

- Observe how each protocol handles coordinator and participant failures.
- Consider blocking vs non-blocking trade-offs.

Timeouts and Retries — Introduction

Concepts

- Network delays and failures can leave transactions in uncertain states.
- Timeouts allow a process to detect unresponsive participants.
- Retries attempt to complete operations before aborting.
- Crucial in distributed transactions to avoid inconsistencies.

Examples

- Database commit retry if coordinator does not respond.
- Payment gateway retries in case of network failure.

Timeout Mechanism

How Timeout Works

- Each participant sets a timer after sending a request.
- If no response is received within the timeout, the participant assumes failure.
- Timeout values must balance between network delays and system responsiveness.

Scenario

- Coordinator does not respond: participants initiate retry or abort.

Retry Strategy

Key Points

- Participants retry operations after timeout or network failure.
- Limit retries to prevent indefinite blocking.
- Use exponential backoff to reduce network congestion.
- Combine retries with logging to ensure reliability.

Practical Example

- Microservices retry API calls during partial failures.

Failure Detection

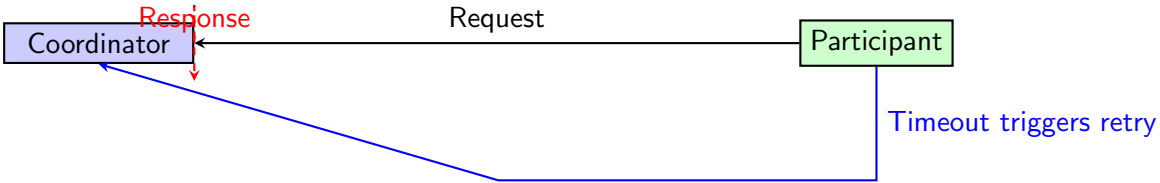
Detecting Failures

- Crash failures: process stops unexpectedly.
- Omission failures: messages lost or delayed.
- Detection via heartbeat messages or acknowledgments.
- Crucial for initiating retries or aborts.

Example

- Distributed database nodes send heartbeat to monitor coordinator.

Illustrative — Timeout Retry



Handling Coordinator Crashes

Mechanisms

- Participants detect unresponsive coordinator via timeout.
- Initiate recovery or abort transaction to avoid inconsistent states.
- Use persistent logs to resume transaction after crash.

Example

- Banking transaction coordinator crashes: participants rollback to maintain consistency.

Retry Limits and Backoff

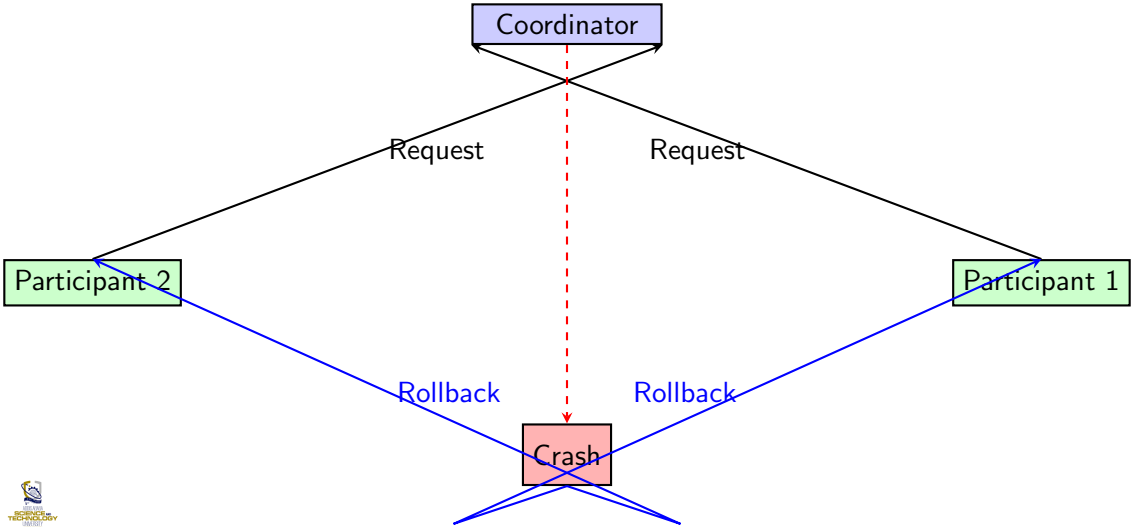
Best Practices

- Limit number of retries to avoid infinite loops.
- Exponential backoff reduces network congestion.
- Combine with logging for reliable recovery.
- Decide abort vs retry based on transaction criticality.

Scenario

- API call retries: 1s, 2s, 4s delays before aborting.

Illustrative— Coordinator Crash



Practical Scenario — Distributed Payment

Example

- Online payment system: multiple nodes validate transaction.
- Coordinator crashes after some nodes vote Yes.
- Timeout detection triggers rollback to maintain consistency.

Lesson

- Reliable transactions require **timeouts, retries, and crash handling**.

Key Takeaways — Timeouts, Retries Failure Handling

Summary

- Timeouts detect unresponsive participants or coordinators.
- Retries attempt completion while limiting network congestion.
- Failure handling ensures rollback or recovery for consistency.
- Proper logging and backoff strategies enhance reliability.

Practical Scenario — Microservices Payment

Example

- A user initiates a payment via multiple microservices (Auth, Billing, Ledger).
- One service fails after updating partial state.
- Timeout triggers retries or rollback to maintain overall consistency.

Lesson

- Distributed systems require **coordinated retries, timeouts, and failure handling**.

Practical Scenario — Blockchain Atomic Commit

Example

- A smart contract transaction is proposed to multiple blockchain nodes.
- Some nodes are temporarily offline.
- Consensus algorithm ensures **either all nodes commit or none commit**, preventing inconsistent state.

Lesson

- **Atomicity and consensus** in distributed ledgers prevent partial failures.

Practical Scenario — Multi-node Order Processing

Example

- An e-commerce order triggers inventory, payment, and shipping microservices.
- One service fails mid-transaction.
- Timeout mechanism triggers **rollback or compensating actions** to maintain consistency.

Lesson

- **Retries and compensating transactions** are essential for reliable distributed operations.

Practical Scenario — Cross-Border Payment System

Example

- Payment transactions span multiple banks and clearing systems.
- Network failure occurs during transaction propagation.
- Coordinators use **timeout and retry mechanisms** to ensure global consistency.

Lesson

- Distributed transaction protocols **guarantee consistency** despite network failures.

Questions?

Thank you for your attention!

References

Suggested References

1. M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., Version 4.02, February 2024.
2. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed., Addison-Wesley, 2012.
3. N. Lynch, *Distributed Algorithms*, 2nd ed., Morgan Kaufmann, 1996.
4. R. E. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 4th ed., Pearson, 2024.