Distributed Transactions: Architecture Guide

Overview

Attribute	Details
Duration	60 minutes
Level	Intermediate to Advanced
Prerequisites	Database fundamentals, microservices basics

Learning Objectives

- Understand why distributed transactions are fundamentally hard
- Compare 2PC, 3PC, Saga, and Eventual Consistency patterns
- Make informed architectural trade-offs for your use case
- Design compensation and recovery strategies
- Handle edge cases and failure scenarios

---

1. The Distributed Transaction Challenge

Why It's Fundamentally Hard

```
Monolithic (Easy):

             Single Database
  BEGIN TRANSACTION
    UPDATE accounts SET balance -= 100
    UPDATE accounts SET balance += 100
  COMMIT

         ↑ ACID guaranteed by DB

Distributed (Hard):

  Order         Payment       Inventory
   DB1           DB2          DB3

         ↑ No single authority to guarantee ACID
```

Core Questions: - What if Payment succeeds but Inventory fails? - What if network partitions during commit? - What if a service crashes mid-transaction?

CAP Theorem: The Fundamental Trade-off

```
flowchart TD
    subgraph CAP["CAP: Pick Two"]
        direction TB
        C["C: Consistency"] --- A["A: Availability"]
        A --- P["P: Partition Tolerance"]
        P --- C
    end

    C -.- CP["CP: 2PC, Locks"]
    A -.- CA["CA: Single DB"]
    P -.- AP["AP: Saga"]
```

Choice	What You Get	What You Sacrifice	Example
CP	Strong consistency	Availability during partitions	2PC, distributed locks
AP	High availability	Immediate consistency	Saga, eventual consistency
CA	Both C and A	Not truly distributed	Single PostgreSQL

Reality: Network partitions WILL happen → You must choose between C and A

---

2. Two-Phase Commit (2PC)

How It Works

```
sequenceDiagram
    participant CO as Coordinator
    participant A as Participant A
    participant B as Participant B
    participant C as Participant C

    rect rgb(230, 240, 255)
    Note over CO,C: Phase 1: PREPARE (Voting)
    CO->>A: PREPARE
    CO->>B: PREPARE
    CO->>C: PREPARE
    A-->>CO: YES (locks held)
    B-->>CO: YES (locks held)
    C-->>CO: YES (locks held)
    end

    Note over CO: All YES → Decision: COMMIT

    rect rgb(220, 255, 220)
    Note over CO,C: Phase 2: COMMIT
    CO->>A: COMMIT
    CO->>B: COMMIT
    CO->>C: COMMIT
    A-->>CO: ACK (locks released)
    B-->>CO: ACK (locks released)
    C-->>CO: ACK (locks released)
    end
```

2PC State Machine

```
Coordinator States:
    INIT → WAITING → COMMITTED/ABORTED

Participant States:
    INIT → PREPARED → COMMITTED/ABORTED
             ↑
         Locks held here (blocking!)
```

Critical Problems with 2PC

Problem	Scenario	Impact
Blocking	Participants hold locks during voting	Throughput drops, deadlock risk
Coordinator SPOF	Coordinator crashes after PREPARE	Participants stuck indefinitely
Network Partition	Can't reach all participants	Transaction hangs
Latency	2 round trips minimum	Not suitable for high-frequency

Edge Cases & Failure Scenarios

```
Scenario 1: Coordinator fails after sending PREPARE

  Coordinator: PREPARE sent → CRASH
  Participant A: PREPARED (holding locks)
  Participant B: PREPARED (holding locks)

  Result: Both participants BLOCKED indefinitely
  Solution: Timeout + new coordinator election
            But may cause inconsistency!
```

```
Scenario 2: Participant fails after voting YES

  Participant A: YES → CRASH → RECOVERS

  On recovery, must:
  1. Check transaction log
  2. Ask coordinator for decision
  3. If coordinator also crashed → UNCERTAIN STATE
```

```
Scenario 3: Network partition during Phase 2

  Coordinator: Sends COMMIT
  Participant A: Receives COMMIT
  Participant B: Network timeout

  Result: A committed, B uncertain
  Must retry COMMIT to B until success
```

When to Use 2PC

Good Fit	Poor Fit
Financial systems requiring strong consistency	High-throughput systems
Small number of participants (2-3)	Many microservices
Low-latency network (same datacenter)	Cross-region deployments
Batch processing jobs	User-facing real-time APIs

3. Three-Phase Commit (3PC)

Improvement Over 2PC

```
sequenceDiagram
    participant CO as Coordinator
    participant P as Participants

    rect rgb(230, 240, 255)
    Note over CO,P: Phase 1: CAN-COMMIT
    CO->>P: Can you commit?
    P-->>CO: YES/NO
    end

    rect rgb(255, 245, 220)
    Note over CO,P: Phase 2: PRE-COMMIT
    CO->>P: Prepare to commit
    P-->>CO: ACK
    Note over P: Key: Can commit on timeout!
    end

    rect rgb(220, 255, 220)
    Note over CO,P: Phase 3: DO-COMMIT
    CO->>P: Commit now
    P-->>CO: Done
    end
```

3PC vs 2PC Trade-offs

Aspect	2PC	3PC
Phases	2	3
Blocking on coordinator failure	Yes (indefinite)	No (timeout-based recovery)
Latency	Lower (2 RTT)	Higher (3 RTT)
Complexity	Medium	High
Network partition safety	Problematic	Still problematic
Practical adoption	Common (XA)	Rare

Why 3PC Isn't Widely Used

```
Problem: 3PC still fails under network partitions

Scenario: Network splits during PRE-COMMIT

  Partition A: Coordinator + Participant 1
  Partition B: Participant 2, 3

  Partition A: Times out → COMMIT (has majority?)
  Partition B: Times out → COMMIT (assumed safe)

  Result: Both partitions may make different
          decisions → INCONSISTENCY
```

```
Reality: Network partitions are common in distributed systems
         → 3PC doesn't solve the fundamental problem
         → Industry moved to Saga/Eventual Consistency instead
```

---

4. Saga Pattern

Core Concept: Local Transactions + Compensation

```
flowchart LR
    subgraph forward["Forward Flow (Happy Path)"]
        T1["T1: Create Order"] --> T2["T2: Process Payment"]
        T2 --> T3["T3: Reserve Inventory"]
        T3 --> T4["T4: Arrange Shipping"]
    end
```

```
flowchart RL
    subgraph compensation["Compensation Flow (T3 Fails)"]
        F["T3 Failed  "] --> C2["C2: Refund Payment"]
        C2 --> C1["C1: Cancel Order"]
    end
```

Key Principle

```
Each step Ti has a compensating transaction Ci

If Tn fails:
  Execute Cn-1, Cn-2, ... C1 in reverse order

Important: Compensation   Rollback
  - Rollback: Undo as if never happened
  - Compensation: Apply corrective action (visible in history)
```

Choreography vs Orchestration

```
flowchart TB
    subgraph choreo["Choreography: Event-Driven"]
        O1["Order Service"] -->|"OrderCreated"| E1["Event Bus"]
        E1 -->|"OrderCreated"| P1["Payment Service"]
        P1 -->|"PaymentDone"| E1
        E1 -->|"PaymentDone"| I1["Inventory Service"]
    end
```

```
flowchart TB
    subgraph orch["Orchestration: Central Control"]
        ORCH["Saga Orchestrator"]
        ORCH -->|"1. CreateOrder"| O2["Order Service"]
        ORCH -->|"2. ProcessPayment"| P2["Payment Service"]
        ORCH -->|"3. ReserveInventory"| I2["Inventory Service"]
    end
```

Choreography vs Orchestration Trade-offs

Aspect	Choreography	Orchestration
Coupling	Loose	Tighter
Single Point of Failure	No	Yes (orchestrator)
Visibility	Hard to track flow	Easy to monitor
Debugging	Difficult	Straightforward
Adding new steps	Modify multiple services	Modify orchestrator only
Cyclic dependencies	Risk of event loops	Not possible
Team autonomy	High	Lower

Architecture Decision Guide

```
Choose CHOREOGRAPHY when:
  Teams are autonomous and own their services
  Flow is simple (< 4 steps)
  Services are truly independent
  You have good distributed tracing

Choose ORCHESTRATION when:
  Flow is complex (> 4 steps)
  Business logic is centralized
  You need clear visibility/monitoring
  Compensation logic is complex
  Regulatory/audit requirements exist
```

Critical Edge Cases

Edge Case 1: Compensation Fails

```
Scenario: Payment refund fails during compensation

  T1: Order Created
  T2: Payment Processed
  T3: Inventory Reserve FAILED
  C2: Refund Payment FAILED   ← What now?

  Solutions:
  1. Retry with exponential backoff
  2. Dead letter queue for manual intervention
  3. Scheduled reconciliation job
  4. Alert operations team
```

Edge Case 2: Duplicate Execution

```
Scenario: Network timeout, message redelivered

  Payment Service receives "ProcessPayment" twice

  Without idempotency:
  → Customer charged twice!

  Solution: Idempotency keys
  1. Each request has unique idempotency_key
  2. Store processed keys in database
```

```
   3. Check before processing, skip if exists
```

Edge Case 3: Out-of-Order Events

```
Scenario: Events arrive out of order

   Expected: OrderCreated → PaymentDone → Shipped
   Actual:   PaymentDone → OrderCreated → Shipped

   Solutions:
   1. Event versioning/sequencing
   2. State machine validation
   3. Buffer and reorder
   4. Reject and retry later
```

Edge Case 4: Long-Running Transactions

```
Scenario: Shipping takes 3 days

   Problem: Can't hold resources for days

   Solutions:
   1. Reservation pattern (soft lock with expiry)
   2. Split into sub-sagas
   3. State machine with timeout transitions
   4. Async notification when complete
```

Saga State Machine Design

```
stateDiagram-v2
    [*] --> STARTED
    STARTED --> ORDER_CREATED: createOrder()
    ORDER_CREATED --> PAYMENT_PROCESSED: processPayment()
    PAYMENT_PROCESSED --> INVENTORY_RESERVED: reserveInventory()
    INVENTORY_RESERVED --> COMPLETED: success

    ORDER_CREATED --> COMPENSATING: failure
    PAYMENT_PROCESSED --> COMPENSATING: failure
    INVENTORY_RESERVED --> COMPENSATING: failure

    COMPENSATING --> COMPENSATED: all compensations done
    COMPENSATING --> COMPENSATION_FAILED: compensation fails

    COMPLETED --> [*]
    COMPENSATED --> [*]
    COMPENSATION_FAILED --> MANUAL_INTERVENTION
```

---

5. Eventual Consistency & Outbox Pattern

The Dual Write Problem

```
flowchart LR
    S["Service"] -->|"1. Write DB  "| DB[(Database)]
    S -->|"2. Publish Event  "| MQ[Message Broker]

    style MQ stroke:#ff0000,stroke-width:2px
```

```
Problem: Two separate systems, no atomic guarantee

Failure scenarios:
1. DB write succeeds, event publish fails
   → Data saved but other services never notified

2. Event published, DB write fails
   → Other services act on non-existent data

3. Service crashes between the two operations
   → Inconsistent state
```

Transactional Outbox Pattern

```
flowchart TB
    S["Service"] --> TX

    subgraph TX["Single Database Transaction"]
        W1["1. Write business data"]
        W2["2. Write event to outbox table"]
    end

    TX --> DB[(Database)]

    RELAY["Message Relay<br/>(Polling or CDC)"] -->|"Read outbox"| DB
    RELAY -->|"Publish"| KAFKA["Message Broker"]
```

Outbox Table Design

```sql
CREATE TABLE outbox_events (
    id              UUID PRIMARY KEY,
    aggregate_type  VARCHAR(255),    -- e.g., 'Order'
    aggregate_id    VARCHAR(255),    -- e.g., order_id
    event_type      VARCHAR(255),    -- e.g., 'OrderCreated'
    payload         JSONB,           -- event data
    created_at      TIMESTAMP,
    published_at    TIMESTAMP NULL,  -- NULL = not yet published

    INDEX idx_unpublished (published_at) WHERE published_at IS NULL
);
```

Message Relay Strategies

Strategy	Pros	Cons
Polling	Simple, no extra infrastructure	Latency, DB load
CDC (Debezium)	Real-time, low DB load	Complex setup
Transaction log tailing	Very efficient	DB-specific

CDC vs Polling Trade-offs

```
Polling:
  Latency: 1-5 seconds (configurable)
  DB Load: Constant queries
  Complexity: Low
  Ordering: Must handle carefully
  Best for: Simple setups, low volume

CDC (Change Data Capture):
  Latency: Milliseconds
  DB Load: Minimal (reads transaction log)
  Complexity: High (Kafka Connect, Debezium)
  Ordering: Guaranteed by log position
  Best for: High volume, low latency requirements
```

Idempotent Consumer Pattern

```
Problem: Message broker may deliver same message twice
         (at-least-once delivery)

Solution: Track processed message IDs


  Consumer receives message
      ↓
  Check: Is message_id in processed_messages table?
      ↓
  YES → Skip (already processed)
  NO  → Process + Insert message_id + Commit

  All in single transaction!
```

---

6. Pattern Comparison & Selection Guide

Comprehensive Comparison

Aspect	2PC	3PC	Saga	Eventual Consistency
Consistency	Strong	Strong	Eventual	Eventual
Isolation	Full	Full	None	None
Availability	Low	Medium	High	High
Latency	High	Higher	Medium	Low
Scalability	Poor	Poor	Good	Excellent
Complexity	Medium	High	High	Medium

Aspect	2PC	3PC	Saga	Eventual Consistency
Recovery	Automatic	Timeout-based	Compensation	Retry + Idempotency

Decision Matrix

```
flowchart TB
    START["Need distributed transaction?"] --> Q1{"Strong consistency<br/>required?"}

    Q1 -->|"Yes"| Q2{"Can tolerate<br/>blocking?"}
    Q1 -->|"No"| Q3{"Complex multi-step<br/>workflow?"}

    Q2 -->|"Yes"| TWO_PC["2PC<br/>(XA Transactions)"]
    Q2 -->|"No"| CONSIDER["Consider relaxing<br/>consistency requirements"]

    Q3 -->|"Yes"| SAGA["Saga Pattern"]
    Q3 -->|"No"| OUTBOX["Outbox + Eventual<br/>Consistency"]

    SAGA --> Q4{"Need visibility<br/>& control?"}
    Q4 -->|"Yes"| ORCH["Orchestration"]
    Q4 -->|"No"| CHOREO["Choreography"]
```

Industry Use Cases

Company/Domain	Pattern	Reason
Banking (transfers)	2PC or Saga with strict compensation	Regulatory, money involved
E-commerce (orders)	Saga (orchestration)	Complex flow, need visibility
Social media (posts)	Eventual consistency	High scale, consistency less critical
Ride-sharing (booking)	Saga (choreography)	Real-time, multiple services
Inventory systems	Saga + reservation pattern	Prevent overselling

---

7. Production Considerations

Monitoring & Observability

```
Essential Metrics:
  Saga completion rate
  Compensation frequency
  Average saga duration
  Failed/stuck sagas count
  Outbox lag (unpublished events)
  Message processing latency

Essential Logs:
  Saga state transitions
  Compensation triggers
  Retry attempts
  Timeout events

Distributed Tracing:
  Correlation ID across all services
  Span for each saga step
  Parent-child relationship for compensation
```

Failure Recovery Strategies

```
Strategy 1: Automatic Retry
  Exponential backoff: 1s, 2s, 4s, 8s...
  Max retries: 3-5 typically
  Circuit breaker after threshold
  Alert on repeated failures

Strategy 2: Dead Letter Queue
  Move failed messages to DLQ
  Manual inspection and replay
  Audit trail preserved
  No blocking of other messages

Strategy 3: Scheduled Reconciliation
  Periodic job compares expected vs actual state
  Fixes inconsistencies automatically
  Reports discrepancies
  Last resort safety net
```

Testing Distributed Transactions

```
Test Categories:
  Happy path (all services succeed)
  Single service failure
  Multiple service failures
  Network partition simulation
  Timeout scenarios
  Duplicate message handling
  Out-of-order message handling
  Compensation failure scenarios

Tools:
  Chaos engineering (Chaos Monkey, Litmus)
  Network fault injection (Toxiproxy)
  Contract testing (Pact)
  Integration test containers
```

---

8. Key Takeaways

#	Takeaway
1	**2PC provides strong consistency** but blocks and doesn't scale
2	**3PC reduces blocking** but still fails under network partitions
3	**Sagas trade isolation for availability** — design for compensation

#	Takeaway
4	**Choreography is loosely coupled** but hard to debug
5	**Orchestration centralizes logic** but creates SPOF
6	**Outbox pattern solves dual-write** — use CDC for production
7	**Design for idempotency** — messages will be delivered multiple times
8	**Compensation Rollback** — it's a corrective action, not undo
9	**Monitor saga health** — stuck sagas indicate systemic issues
10	**Choose based on requirements** — not all systems need strong consistency

---

9. Practical Exercise

Design Challenge

Design a distributed transaction strategy for a ride-sharing booking:

Flow: 1. User requests ride 2. Find available driver 3. Reserve driver (can't accept other rides) 4. Process payment authorization 5. Confirm booking

Requirements: - Driver reservation expires after 30 seconds - Payment failures should release driver - Handle driver cancellation after booking - Support concurrent booking attempts for same driver

Discussion Questions: 1. Choreography or orchestration? Why? 2. How do you handle "driver reserved but payment timeout"? 3. What if compensation (release driver) fails? 4. How do you prevent double-booking a driver? 5. How do you show booking status to user in real-time?