2-Phase Commit Protocol: Complete Guide

What is 2-Phase Commit (2PC)?

2-Phase Commit is a distributed algorithm that ensures all participants in a transaction either **commit together** or **abort together**. It prevents partial failures that leave systems in inconsistent states.

Core Principle: *"Either everyone succeeds, or everyone fails - no partial updates."*

The Problem It Solves

Without 2PC (Disaster Scenario):

```
Bank Transfer: $100 from Account A to Account B
Step 1: Deduct $100 from Account A ☑ Success
Step 2: Add $100 to Account B     ❌ Network failure!
Result: Money vanished! Account A: -$100, Account B: +$0
```

With 2PC (Safe Operation):

Phase 1 - Ask Permission:

```
Phase 1 - Ask Permission:
- Account A: "Can you deduct $100?" → ☑ "Yes"
- Account B: "Can you add $100?"    → ☑ "Yes"

Phase 2 - Execute Together:
- Account A: Actually deduct $100   → ☑ Done
- Account B: Actually add $100      → ☑ Done
Result: Both succeed! Account A: -$100, Account B: +$100
```

Key Components

Component	Role	Responsibility
Coordinator	Transaction Manager	Orchestrates entire process, makes final decision
Participants	Resource Managers	Individual services/databases involved
Transaction Log	Recovery System	Stores transaction state for failure recovery

Phase 1: PREPARE (Voting Phase)

Coordinator Actions:

1. Send PREPARE messages to all participants
2. Wait for votes from all participants
3. Decision Logic:
    - ALL vote YES → Proceed to commit
    - ANY vote NO → Proceed to abort

Participant Actions:

1. Receive PREPARE message
2. Check if operation possible:
    - Verify resources (balance, stock, etc.)
    - Check constraints and locks
    - Prepare changes (but don't commit yet)
3. Vote YES or NO based on feasibility
4. If YES: Lock resources and wait for decision

Example: E-commerce Order

```
// Customer wants to buy 1 Laptop ($800)
PREPARE Phase:
☑ User Service: "Balance $1000 >= $800? YES"
☑ Inventory: "Stock 5 >= 1? YES"
☑ Order Service: "Can create record? YES"
→ All voted YES, proceed to Phase 2
```

Phase 2: COMMIT/ABORT (Decision Phase)

If ALL Voted YES (COMMIT):

Coordinator: Sends COMMIT to all participants Participants: Actually execute their operations

```
☑ User Service: Deduct $800 (Balance: $1000 → $200)
☑ Inventory: Reduce stock by 1 (Stock: 5 → 4)
☑ Order Service: Create order record
→ Transaction completed successfully
```

If ANY Voted NO (ABORT):

Coordinator: Sends ABORT to all participants Participants: Release locks, rollback prepared changes

```
❌ User Service: "Insufficient balance $500 < $800"
→ ABORT sent to all participants
→ No changes made, system remains consistent
```

ACID Properties Guaranteed

Property	2PC Implementation	Example
Atomicity	All operations succeed or all fail	Money transfer: both debit AND credit happen
Consistency	Prepare phase validates constraints	No negative balances, no overselling
Isolation	Resources locked during transaction	Two customers can't buy last item
Durability	Changes written to persistent storage	Survives system crashes

Real-World Examples

Banking Transfer

```
Transaction: Transfer $500 from Account A to B
Phase 1: Both accounts confirm they can handle the operation
Phase 2: Both accounts execute simultaneously
Result: A: $1000→$500, B: $200→$700 (Total preserved)
```

Microservices Order

```
Services: Payment, Restaurant, Delivery, Inventory
Phase 1: All services confirm capability
  - Payment: "Card valid, can charge $25"
  - Restaurant: "Can prepare pizza in 20 min"
  - Delivery: "Driver available"
  - Inventory: "Ingredients in stock"
Phase 2: All services execute their operations
```

Advantages vs Disadvantages

✅ Advantages

- Guaranteed Atomicity: No partial failures
- Strong Consistency: All participants agree
- Proven Reliability: Used by banks for decades
- Wide Support: Most enterprise databases support it

❌ Disadvantages

- Blocking Protocol: Participants wait for coordinator
- Single Point of Failure: Coordinator dependency
- High Latency: Multiple network round-trips
- Resource Locking: Reduces system concurrency

Common Failure Scenarios & Solutions

Failure	Impact	Solution
Coordinator crashes before Phase 2	Participants blocked	Timeout mechanisms, coordinator recovery
Participant fails during Phase 1	Coordinator waits indefinitely	Assume NO vote on timeout
Network partition	Some participants unreachable	Use consensus protocols
Coordinator crashes after COMMIT	Some participants don't know result	Re-send messages on restart

Implementation

1. 📌 Project Overview

This project demonstrates the Two-Phase Commit (2PC) protocol using a simplified E-commerce application.
It simulates a distributed transaction where multiple participants (Users, Products, Orders) must either all commit or all abort, ensuring atomicity and consistency in a transaction.

The project uses:

- Backend: Node.js + Express
- Database: MongoDB (standalone, no replica set needed)
- Frontend: Static HTML/JS (for testing via forms/buttons)

Use case:
When a user purchases a product → system checks balance & stock (Phase 1) → then deducts balance, reduces stock, and creates order (Phase 2).
If any step fails → the transaction is aborted safely.

2. 📂 Project Structure

ecommerce-2pc/

```
├── package.json          # Project dependencies & scripts
├── package-lock.json     # Dependency lock file
├── server.js             # Main backend server (2PC implementation)
│
├── public/               # Frontend (static files served by Express)
│   └── index.html        # UI for testing transactions
│
└── README.md             # Documentation (this file)
```

3. ⚙️ Setup & Installation

Prerequisites

- Node.js (>= 16.x)
- MongoDB (standalone instance running locally on mongodb://localhost:27017)

Steps

1. Clone repo

   git clone <repo-url>

   cd

2. # Install dependencies

   npm install

3. # Start server

   node server.js

4. Access

- Backend API → http://localhost:3000/api/...
- Frontend UI → http://localhost:3000/

5. 🔄 Two-Phase Commit Flow

Phase 1: Prepare

1. Validate user exists.
2. Validate product exists.
3. Check sufficient balance.
4. Check sufficient stock.
5. Insert transaction record in transactions collection with status PREPARED.

👉 If any step fails → transaction aborted.

Phase 2: Commit

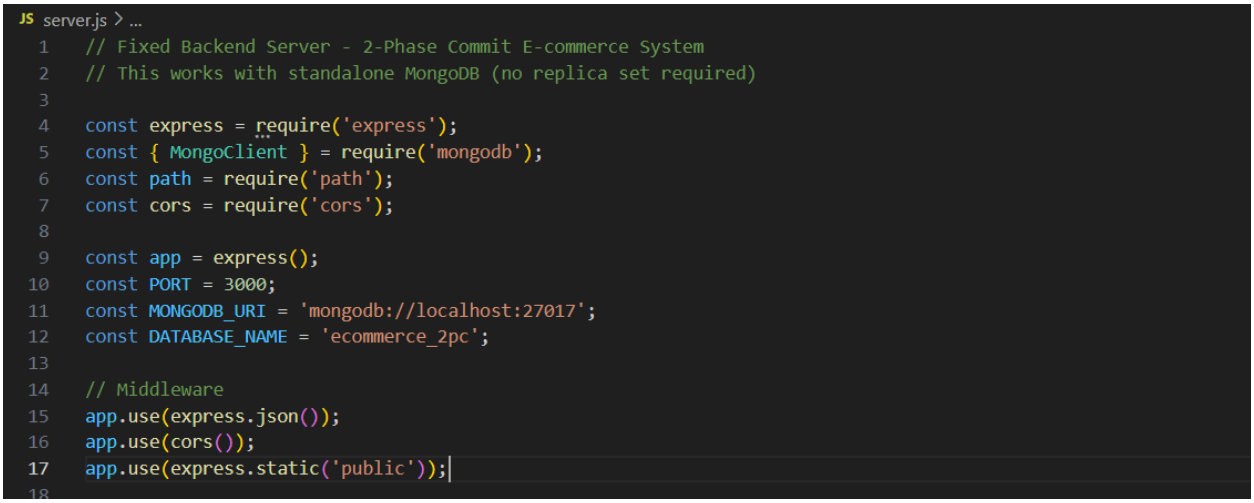
1. Deduct user balance.
2. Reduce product stock.
3. Insert new order into orders collection.
4. Update transaction status to COMMITTED.

👉 If any step fails → rollback & mark as ABORTED.

6.Code Explanation

Server.js

Let's see server.js code with detailed explanation.



```
server.js > ...
1   // Fixed Backend Server - 2-Phase Commit E-commerce System
2   // This works with standalone MongoDB (no replica set required)
3
4   const express = require('express');
5   const { MongoClient } = require('mongodb');
6   const path = require('path');
7   const cors = require('cors');
8
9   const app = express();
10  const PORT = 3000;
11  const MONGODB_URI = 'mongodb://localhost:27017';
12  const DATABASE_NAME = 'ecommerce_2pc';
13
14  // Middleware
15  app.use(express.json());
16  app.use(cors());
17  app.use(express.static('public'));
18
```

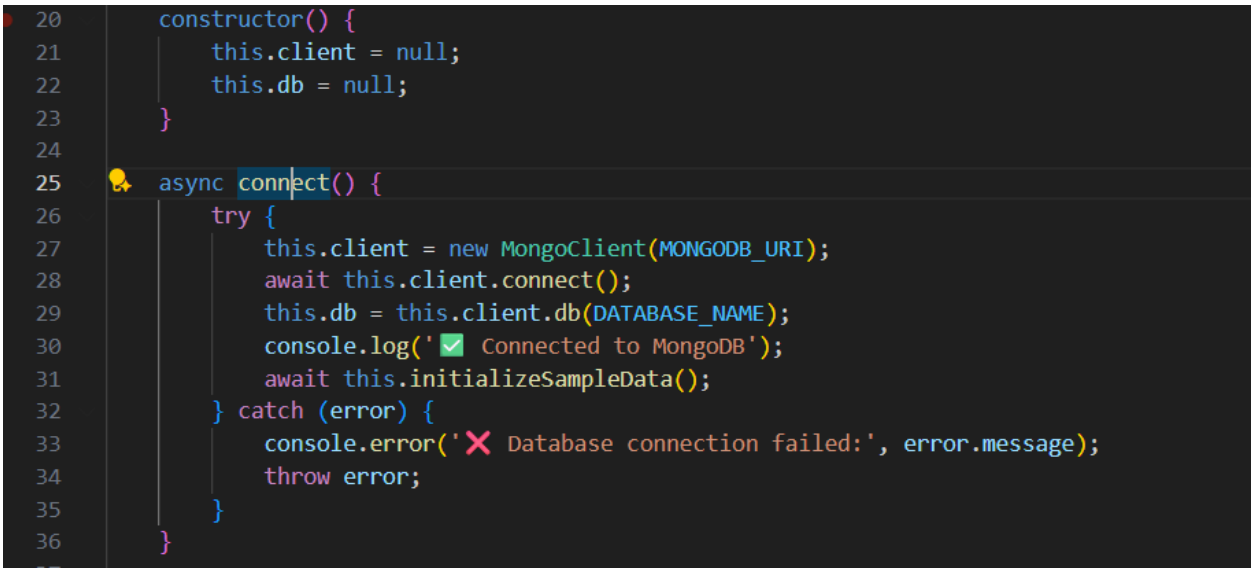
First of all, we will import all the required packaes.

- **express** → to build the web server.
- **mongodb** → to connect to MongoDB.
- **path** → to handle file paths.
- **cors** → to allow requests from other origins (like frontend-backend).

Then I have initialized Express Object. And defined Serer Port and MongoDB URL with database name.

- app → the main Express application object.
- PORT → the server will run on port **3000**.
- MONGODB_URI → the MongoDB connection URL (local database).
- DATABASE_NAME → the name of the database (ecommerce_2pc).

Then I have created a class TwoPhaseCommitSystem. Which will going to have methods to perform two phase communications.



```
20    constructor() {
21        this.client = null;
22        this.db = null;
23    }
24
25    async connect() {
26        try {
27            this.client = new MongoClient(MONGODB_URI);
28            await this.client.connect();
29            this.db = this.client.db(DATABASE_NAME);
30            console.log('✅ Connected to MongoDB');
31            await this.initializeSampleData();
32        } catch (error) {
33            console.error('❌ Database connection failed:', error.message);
34            throw error;
35        }
36    }
```

constructor() will runs automatically when the class is created.

Inside it:

- this.client = null; → will later store the MongoDB connection client.
- this.db = null; → will later store the database reference.

This class has two properties and one connect method.

Connect method then initialize new MongoDB client Object.

Think of it like:

- 🚌 A **driver** that knows how to talk to the MongoDB database.
- It is not the database itself — it's the **connection tool** that lets your Node.js app send and receive data from MongoDB.

And then "**await this.client.connect()**" will Actually **connects** to MongoDB. (Since it may take time, await is used.) and we will select the database 'ecommerce_2pc'.

And then we will call initializeSampleData method of this class.

```
async initializeSampleData() {
    const users = this.db.collection('users');
    const products = this.db.collection('products');
    const orders = this.db.collection('orders');
    const transactions = this.db.collection('transactions');

    // Clear existing data
    await Promise.all([
        users.deleteMany({}),
        products.deleteMany({}),
        orders.deleteMany({}),
        transactions.deleteMany({})
    ]);

    // Insert sample users
    await users.insertMany([
        { _id: 'user1', name: 'John Doe', balance: 1000 },
        { _id: 'user2', name: 'Jane Smith', balance: 500 },
        { _id: 'user3', name: 'Mike Johnson', balance: 200 }
    ]);

    // Insert sample products
    await products.insertMany([
        { _id: 'prod1', name: 'Laptop', price: 800, stock: 5 },
        { _id: 'prod2', name: 'Mouse', price: 25, stock: 10 },
        { _id: 'prod3', name: 'Keyboard', price: 75, stock: 8 },
        { _id: 'prod4', name: 'Monitor', price: 300, stock: 3 }
    ]);

    console.log('📦 Sample data initialized');
}
```

This method will simply create 4 collections. And insert sample data into those collections.

Now I have created two methods for 2 phases.

One is prepareTransaction

```
// PHASE 1: PREPARE - Check if all operations can be performed
async prepareTransaction(userId, productId, quantity) {
    const transactionId = `txn_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
    const steps = [];

    try {
        steps.push({ step: 'PHASE_1_START', message: '🔍 Starting Phase 1: PREPARE', success: true });

        // Get current data (simulating locks in a real system)
        const users = this.db.collection('users');
        const products = this.db.collection('products');
        const transactions = this.db.collection('transactions');

        // PARTICIPANT 1: Check user exists and has sufficient balance
        const user = await users.findOne({ _id: userId });
        if (!user) {
            throw new Error(`User ${userId} not found`);
        }
        steps.push({ step: 'USER_CHECK', message: `✅ User ${user.name} found`, success: true });

        // PARTICIPANT 2: Check product exists and has sufficient stock
        const product = await products.findOne({ _id: productId });
        if (!product) {
            throw new Error(`Product ${productId} not found`);
        }
        steps.push({ step: 'PRODUCT_CHECK', message: `✅ Product ${product.name} found`, success: true });

        const totalCost = product.price * quantity;
```

```
            // PARTICIPANT 1: Balance Check
            if (user.balance < totalCost) {
                throw new Error(`Insufficient balance. Required: $${totalCost}, Available: $${user.balance}`);
            }
            steps.push({
                step: 'BALANCE_CHECK',
                message: `✅ Balance check passed: $${user.balance} >= $${totalCost}`,
                success: true
            });

            // PARTICIPANT 2: Stock Check
            if (product.stock < quantity) {
                throw new Error(`Insufficient stock. Required: ${quantity}, Available: ${product.stock}`);
            }
            steps.push({
                step: 'STOCK_CHECK',
                message: `✅ Stock check passed: ${product.stock} >= ${quantity}`,
                success: true
            });

            // PARTICIPANT 3: Create pending transaction record
            const transactionRecord = {
                _id: transactionId,
                userId,
                productId,
                quantity,
                totalCost,
                status: 'PREPARED',
                preparedAt: new Date(),
                expiresAt: new Date(Date.now() + 300000), // 5 minutes timeout
                userSnapshot: { balance: user.balance },
                productSnapshot: { stock: product.stock }
            };
```

```
            await transactions.insertOne(transactionRecord);
            steps.push({
                step: 'TRANSACTION_PREPARED',
                message: `✅ Transaction ${transactionId} prepared successfully`,
                success: true
            });

            steps.push({ step: 'PHASE_1_COMPLETE', msesage: '🎉 Phase 1 COMPLETE - All participants voted YES!', success: true });

            return { success: true, transactionId, steps, phase: 1 };

        } catch (error) {
            steps.push({
                step: 'PHASE_1_FAILED',
                message: `❌ Phase 1 FAILED: ${error.message}`,
                success: false
            });

            return { success: false, error: error.message, steps, phase: 1 };
        }
    }

```

In this function I will simply go through below steps.

1. **Start the preparation**
    - Marks the beginning of Phase 1.
    - Keeps track of all steps for logging.
2. **Check if the user exists**
    - Looks up the user in the database.
    - If user doesn't exist → transaction fails immediately.
3. **Check if the product exists**
    - Looks up the product in the database.

- If product doesn’t exist → transaction fails.

4. **Check user balance**
    - Calculates total cost of purchase.
    - Ensures user has enough money.
    - If balance is insufficient → transaction fails.
5. **Check product stock**
    - Ensures the product has enough quantity in stock.
    - If stock is insufficient → transaction fails.
6. **Create a pending transaction record**
    - Saves a “PREPARED” transaction in the database.
    - Stores snapshots of user balance and product stock (for safety).
7. **Return result**
    - If all checks passed → transaction is ready to comm

Second is commitTransaction

```
// PHASE 2: COMMIT - Execute the actual operations
async commitTransaction(transactionId) {
    const steps = [];

    try {
        steps.push({ step: 'PHASE_2_START', message: '🔄 Starting Phase 2: COMMIT', success: true });

        const users = this.db.collection('users');
        const products = this.db.collection('products');
        const orders = this.db.collection('orders');
        const transactions = this.db.collection('transactions');

        // Get the prepared transaction
        const transaction = await transactions.findOne({ _id: transactionId, status: 'PREPARED' });

        if (!transaction) {
            throw new Error(`Transaction ${transactionId} not found or not in PREPARED state`);
        }

        if (new Date() > transaction.expiresAt) {
            throw new Error(`Transaction ${transactionId} has expired`);
        }

        // Double-check that resources haven't changed since prepare
        const currentUser = await users.findOne({ _id: transaction.userId });
        const currentProduct = await products.findOne({ _id: transaction.productId });

        if (currentUser.balance !== transaction.userSnapshot.balance) {
            throw new Error(`User balance changed since prepare phase`);
        }
```

```
        if (currentProduct.stock !== transaction.productSnapshot.stock) {
            throw new Error(`Product stock changed since prepare phase`);
        }

        // PARTICIPANT 1: Deduct user balance
        const userUpdateResult = await users.updateOne(
            { _id: transaction.userId, balance: transaction.userSnapshot.balance }, // Ensure balance hasn't changed
            { $inc: { balance: -transaction.totalCost } }
        );

        if (userUpdateResult.matchedCount === 0) {
            throw new Error(`Failed to update user ${transaction.userId} - balance may have changed`);
        }
        steps.push({
            step: 'BALANCE_DEDUCTED',
            message: `✅ Deducted $${transaction.totalCost} from user balance`,
            success: true
        });

        // PARTICIPANT 2: Reduce product stock
        const productUpdateResult = await products.updateOne(
            { _id: transaction.productId, stock: transaction.productSnapshot.stock }, // Ensure stock hasn't changed
            { $inc: { stock: -transaction.quantity } }
        );
```

```
            if (productUpdateResult.matchedCount === 0) {
                // Rollback user balance
                await users.updateOne(
                    { _id: transaction.userId },
                    { $inc: { balance: transaction.totalCost } }
                );
                throw new Error(`Failed to update product ${transaction.productId} - stock may have changed`);
            }
            steps.push({
                step: 'STOCK_REDUCED',
                message: `✅ Reduced stock by ${transaction.quantity}`,
                success: true
            });

            // PARTICIPANT 3: Create order record
            const orderId = `order_${Date.now()}`;
            await orders.insertOne({
                _id: orderId,
                userId: transaction.userId,
                productId: transaction.productId,
                quantity: transaction.quantity,
                totalCost: transaction.totalCost,
                status: 'COMPLETED',
                createdAt: new Date()
            });
            steps.push({
                step: 'ORDER_CREATED',
                message: `✅ Order ${orderId} created successfully`,
                success: true
            });
```

```
            // Update transaction status to committed
            await transactions.updateOne(
                { _id: transactionId },
                {
                    $set: {
                        status: 'COMMITTED',
                        committedAt: new Date(),
                        orderId: orderId
                    }
                }
            );

            steps.push({ step: 'PHASE_2_COMPLETE', message: '🎉 Phase 2 COMPLETE - Transaction committed successfully!', success: true });

            return { success: true, orderId, steps, phase: 2 };

        } catch (error) {
            steps.push({
                step: 'PHASE_2_FAILED',
                message: `❌ Phase 2 FAILED: ${error.message}`,
                success: false
            });

            // Mark transaction as aborted
            await this.abortTransaction(transactionId);

            return { success: false, error: error.message, steps, phase: 2 };
        }
    }
```

This function will go through below stages.

1. **Start Phase 2**
    - Marks the beginning of the commit phase.
    - Keeps a log of all steps for debugging or UI display.
2. **Get database collections**
    - Access users, products, orders, and transactions collections.
3. **Fetch the prepared transaction**
    - Looks for the transaction by ID and ensures it's in **PREPARED** state.
    - If not found or already expired → transaction fails.

4. **Verify resources haven't changed**
    - Checks that the user's balance and product stock are **same as in Phase 1 snapshot**.
    - If anything changed → transaction fails (prevents inconsistencies).
5. **Deduct user balance**
    - Reduces user balance by the total cost of the order.
    - If deduction fails → transaction fails.
6. **Reduce product stock**
    - Decreases product quantity in stock.
    - If this fails → rollback user balance (undo deduction) and abort transaction.
7. **Create order record**
    - Inserts a new order in the orders collection to finalize the purchase.
8. **Update transaction status to COMMITTED**
    - Marks the transaction as successfully completed.
    - Logs commit time and order ID.
9. **Return success or failure**
    - If all steps pass → return success with order ID.
    - If any step fails → return failure and mark transaction **ABORTED**.

If error occurred in any of the phase abort Transaction method will call and Transaction will be terminated.

```
281     async abortTransaction(transactionId) {
282         try {
283             const transactions = this.db.collection('transactions');
284             await transactions.updateOne(
285                 { _id: transactionId },
286                 {
287                     $set: {
288                         status: 'ABORTED',
289                         abortedAt: new Date()
290                     }
291                 }
292             );
293         } catch (error) {
294             console.error('Error aborting transaction:', error.message);
295         }
296     }
297 }
```

Then I have written a function to connect to express server.

```
382 // Start server
383 async function startServer() {
384     try {
385         await system.connect();
386         app.listen(PORT, () => {
387             console.log(`🚀 Server running at http://localhost:${PORT}`);
388             console.log('📱 Open your browser and go to http://localhost:3000');
389             console.log('\n🎯 This version works with standalone MongoDB!');
390             console.log('💡 No replica set required - perfect for learning!');
391         });
392     } catch (error) {
393         console.error('Failed to start server:', error);
394         process.exit(1);
395     }
396 }
397
398 startServer();
```

Conclusion

2-Phase Commit is essential for scenarios requiring **strong consistency** in distributed systems. While it has performance trade-offs, it's the gold standard for critical operations like financial transactions.

Key Takeaway: 2PC trades performance for correctness - use it when you absolutely cannot afford partial failures, but consider lighter alternatives for less critical operations.

Understanding 2PC provides the foundation for all distributed transaction patterns and is crucial for designing reliable distributed systems.