**Distributed Transactions in Distributed Databases**

A **distributed transaction** refers to a transaction that involves multiple databases or resources, typically across different locations or servers, in a distributed system.

DT is a transaction that spans **multiple databases** or **multiple nodes** in a distributed system. The goal is to ensure **atomicity** — either **all parts of the transaction succeed**, or **none do**.

To achieve this, we use protocols like the **Two-Phase Commit (2PC)** or **Three-Phase Commit (3PC)**.

These transactions aim to maintain the **ACID (Atomicity, Consistency, Isolation, and Durability)** properties even when they span multiple nodes or systems, ensuring that the data remains consistent and reliable.

In a distributed database environment, a single transaction might involve several different systems or databases.

For example, one part of the transaction could update a database in New York, while another part updates a database in London. To handle this successfully, we need a way to ensure that all systems involved in the transaction behave correctly, even if there are network issues or failures.

**ACID property in Distributed Transactions is critical:**

1. **Atomicity:** In the context of a distributed transaction, atomicity ensures that either **all** parts of the transaction are committed (i.e., changes are applied), or **none** of them are. If any part of the transaction fails, the entire transaction is rolled back to maintain consistency.
2. **Consistency:**The distributed transaction must move the system from one consistent state to another. This means that if the transaction is successfully completed, the data should be consistent across all the systems involved. If any part fails, the database must return to its original consistent state.
3. **Isolation:**Isolation ensures that the operations of a distributed transaction are not interfered with by other concurrent transactions. This means that each transaction should be processed in isolation, even if multiple transactions are running simultaneously across distributed systems.
4. **Durability:**Durability guarantees that once a distributed transaction has been committed, its results are permanent, even in the case of a system crash or failure.

**Challenges of Distributed Transactions:**

1. **Network Failures:**Since the transaction spans multiple nodes or databases, network failure between these nodes could cause inconsistencies or prevent the transaction from being completed properly.
2. **Partial Failures:**One part of the transaction may succeed, but another part fails due to issues like server crashes, deadlocks, or resource unavailability.
3. **Concurrency Control:**Managing concurrency across different nodes is difficult. Without proper control, transactions could lead to conflicts or dirty reads across distributed systems.
4. **Latency:**Communication between distributed systems might introduce latency, which can affect the performance of transactions.

**HOW TO MANAGE THIS PROBLEM?**

**Protocols for Handling Distributed Transactions:**

To manage distributed transactions effectively, we rely on certain **protocols** that ensure atomicity, consistency, and fault tolerance across the distributed system. The two main protocols used in distributed transactions are

**Two-Phase Commit (2PC)** and **Three-Phase Commit (3PC)**.

**1. Two-Phase Commit (2PC):**

The **Two-Phase Commit (2PC)** protocol is the most widely used method for ensuring the atomicity of distributed transactions. It operates in two phases:

**Phase 1: Prepare Phase (Voting Phase)**

* The transaction coordinator (the central node managing the transaction) sends a "prepare" message to all participant nodes (databases or resources involved).
* Each participant then locks the resources that will be affected by the transaction and responds with a "yes" (if it is ready to commit) or "no" (if it cannot proceed).

**Phase 2: Commit Phase (Decision Phase)**

* If all participants respond with "yes," the coordinator sends a "commit" message, and all participants apply the changes and finalize the transaction.
* If any participant responds with "no," the coordinator sends a "rollback" message, and all participants undo any changes they made during the transaction.

**Drawbacks of 2PC:**

* It is blocking: If a participant crashes after sending a "yes" vote, the transaction cannot proceed because it needs a response from that participant.
* It has a single point of failure at the coordinator.

**2. Three-Phase Commit (3PC):**The **Three-Phase Commit (3PC)** protocol is an improvement over 2PC designed to address some of its shortcomings, especially in terms of non-blocking and fault tolerance. It introduces an additional phase to handle potential failures more gracefully.

**Phase 1: Can Commit Phase (Prepare Phase):**The coordinator sends a "can commit" message to all participants, asking whether they are ready to commit the transaction. Participants respond with "yes" (if they can commit) or "no" (if they cannot).

**Phase 2: Pre-Commit Phase:**If all participants respond with "yes," the coordinator sends a "pre-commit" message. This instructs all participants to prepare to commit the transaction (i.e., the changes are made in a way that can be safely finalized).

**Phase 3: Commit Phase:**Once all participants acknowledge the pre-commit, the coordinator sends a "commit" message, and the transaction is fully committed. If any participant fails to acknowledge the pre-commit, the transaction is aborted.

**Advantages of 3PC:**

* It is non-blocking: It can handle certain types of failures without leaving participants in an uncertain state.
* It ensures that once the "pre-commit" phase is complete, the transaction will either be committed or aborted.

**Drawbacks of 3PC:**

* It is more complex and incurs additional communication overhead compared to 2PC.

**Example Scenario of a Distributed Transaction:**

Imagine an e-commerce system that needs to process an order. The transaction involves two separate databases:

* **Inventory Database**: To deduct the number of items from stock.
* **Payment Database**: To process the payment for the order.

**Distributed Transaction Steps:**

1. **Begin Transaction:** The transaction starts with a request to process the order.
2. **Prepare Phase:** The coordinator asks both the inventory and payment databases if they are ready to proceed. The inventory database locks the relevant product items, and the payment database locks the payment record.
3. **Commit or Rollback:**If both databases respond positively (i.e., they are ready to commit), the coordinator sends a commit message to both, and the order is finalized.

If one of the databases is unable to commit (e.g., insufficient stock or payment failure), the coordinator sends a rollback message to all databases, and any changes made (e.g., stock deduction) are undone.

**Failure Recovery:**If the coordinator fails after sending the "prepare" message but before receiving all responses, the transaction can be rolled back by retrying the "prepare" phase with each participant once the coordinator recovers.

**Example 2 Scenario**

Imagine you have a system with two databases:

* **DB1 (in New York)**: Manages **user accounts**.
* **DB2 (in London)**: Manages **payment processing**.

**The use case:**

A user transfers $100 from their **bank account** in DB1 to their **payment wallet** in DB2.

You want to ensure that **money is deducted** from DB1 **only if** it's **added to DB2** — and vice versa.

**Using Two-Phase Commit Protocol (2PC)**

**Phase 1: Prepare (Voting Phase)**

1. **Transaction Coordinator** starts the distributed transaction.
2. It sends a **"prepare" request** to DB1 and DB2.
3. Both DB1 and DB2 **check** if they can perform their part:

DB1: Can I deduct $100?

DB2: Can I credit $100?

1. If both are ready, they respond with **“Yes”** (vote to commit).
2. They also write the transaction info to a **local log** but don’t apply changes yet.

**Phase 2: Commit (Decision Phase)**

1. If all participants replied **yes**, the coordinator sends a **"commit"** command to all.
2. DB1 deducts $100 and DB2 credits $100.
3. Both databases **confirm** the commit.

**If anything goes wrong (e.g., one says "No"):**

* The coordinator sends a **"rollback"** command.
* Both DBs **undo** any changes.

**TOOLS TO DO ALL THESE THINGS ARE:: practical approach**

* **XA Transactions** in databases like MySQL, PostgreSQL.
* **Distributed transaction coordinators** in systems like:
  + **Java EE (JTA)**
  + **Microsoft DTC**
  + **Apache Kafka (for transactional messaging)**
  + **Google Spanner** or **CockroachDB** (which use more advanced consensus protocols)

**Caveats**

* 2PC is **blocking** — if the coordinator crashes, participants may be left waiting.
* That’s why **3PC** or **Paxos/Raft-based protocols** are used in highly reliable systems.

##########################################################################