RGPVNOTES.IN

Subject Name: **Distributed System**

Subject Code: **IT-6005**

Semester: **6th**

UNIT 4

Distributed Transactions:

A distributed transaction is a type of transaction with two or more engaged network hosts. Generally, hosts provide resources, and a transaction manager is responsible for developing and handling the transaction. Like any other transaction, a distributed transaction should include all four ACID properties (atomicity, consistency, isolation, durability). Given the nature of the work, atomicity is important to ensure an all-or-nothing outcome for the operations bundle (unit of work).

Properties of transactions

The properties of transactions are summarized with the acronym ACID, which stands for Atomic, Consistent, Isolated, and Durable.

1. **Atomic**

Either an entire transaction happens completely or not at all. If the transaction does happen, it happens as a single indivisible action. Other processes cannot see intermediate results. For example, suppose we have a file that is 100 bytes long and a transaction begins appending to it. If other processes read the file, they only see the 100 bytes. At the end of the transaction, the file instantly grows to its new size.

2. **Consistent**

If the system has certain invariants, they must hold after the transaction (although they may be broken within the transaction). For example, in some banking application, the invariant may be that the amount of money before a transaction must equal the amount of money after the transaction. Within the transaction, this invariant may be violated but this is not visible outside the transaction.

3. **Isolated (or serializable)**

If two or more transactions are running at the same time, to each of them and to others, the final result looks as though all transactions ran sequentially in some order.

An order of running transactions is called a schedule. Orders may be interleaved. If no interleaving is done and the transactions are run in some sequential order, they are serialized.

Consider the following three (small) transactions:

begin	begin	begin
x=0	x=0	x=0
x=x+1	x=x+2	x=x+3
end	end	end

Some possible schedules are (with time flowing from left to right):

schedule	execution order						final x	legal?
schedule 1	x=0	x=x+1	x=0	x=x+2	x=0	x=x+3	3	yes
schedule 1	x=0	x=0	x=x+1	x=x+2	x=0	x=x+3	3	yes
schedule 1	x=0	x=0	x=x+1	x=0	x=x+2	x=x+3	5	NO

4. **Durable**

Once a transaction commits, the results are made permanent. No failure after a commit can undo results or cause them to get lost. [Conversely, the results are not permanent until a transaction commits.]

States of transection:

In order to indicate successful completion, Failure and stopping of a transection(shown in fig 4.1), we define different state of transection they are:

1. **Active state:** this is in initial state, transection stays in this state when it is executed.
2. **Partially committed state:** A transection is in this state when it has executed the final statement.
3. Failed: A transection in this state, when there is any error during execution. So after discovery that normal execution can no longer possible transection comes in this state.

4. **Abort:** A transection is said aborted if after failure it is not possible to go to any other state. The only option then is to kill it and restart. Therefore a transaction is said aborted when transection is rolled back and database is being restored to consistent state prior to the start of the transaction.
5. **Committed:** a transaction is in committed state once it has been successfully executed and database is transformed in new consistent state.

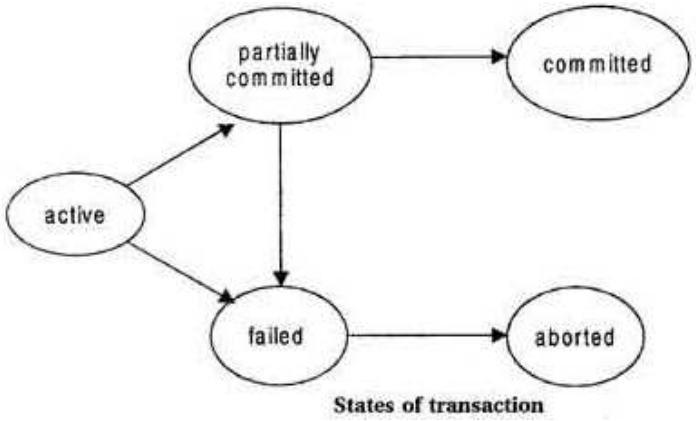

Fig 4.1 states of transections

Flat and nested distributed transactions

A distributed transaction is database transaction in which two or more network hosts are involved. Mostly, hosts provide transactional resources, while transaction manager is responsible for creating and managing a global transaction that encompasses all operations against such resources. Any other transactions must have all four ACID (atomicity, consistency, isolation, durability) properties.

A client transaction becomes distributed if it invokes in several servers.

- **Flat transaction:** client request to more than one server. T invokes operation in objects in 3 server x, y, z so T is flat transaction. A flat client transaction completes each its request before going on the next one. Therefore each transaction accesses servers object sequentially. When servers use locking, a transaction can only be waiting for one object at a time.
- **Nested transactions:** Sub transaction at some level can be run concurrently. It gives better performance than in simple transaction.

In database management systems and operating systems, transactions are used as units of consistency, serializability, recovery, and for deadlock control. Normally, the transactions for each of these systems are considered independently. In this scenario we describe nested transactions where the transactions from one system interact with the transactions from another system. Such nested transactions can expect to become more important with the introduction of network operating systems and heterogeneous distributed database systems. shown in fig 4.2.

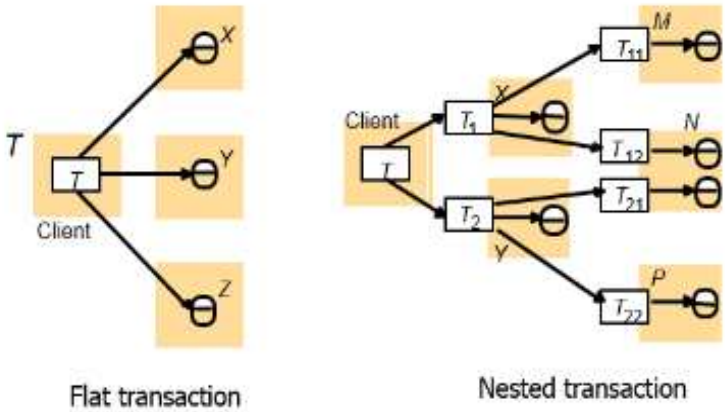

Fig 4.2 Flat and nested distributed transactions

Nested banking transaction:

consider distributed transaction in which a client transfer 10rs from account A to C and then transfer 20rs from B to D. Account A and B are on separate servers X and Y and accounts C and D are at server Z. if this transaction is structured as a set of four nested transactions, as shown in figure, the four requests (two deposit and two withdraw) can run in parallel and overall effect can be achieved with better performance than a simple transaction in which the four operations are invoked sequentially. shown in fig 4.3.

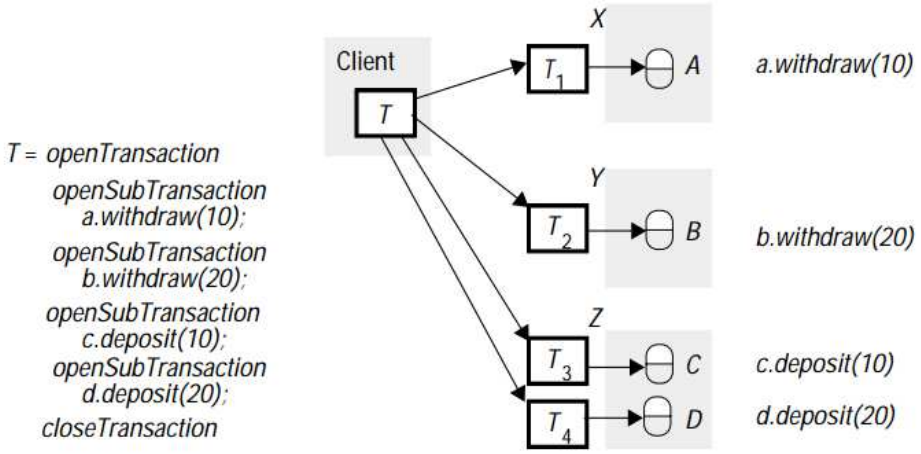

Fig 4.3 Nested banking transaction

Atomic Commit protocols:

In the field of computer science, an atomic commit is an operation that applies a set of distinct changes as a single operation. If the changes are applied then the atomic commit is said to have succeeded. If there is a failure before the atomic commit can be completed then all of the changes completed in the atomic commit are reversed. This ensures that the system is always left in a consistent state. The other key property of isolation comes from their nature as atomic operations. Isolation ensures that only one atomic commit is processed at a time. The most common uses of atomic commits are in database systems and version control systems.

The problem with atomic commits is that they require coordination between multiple systems. As computer networks are unreliable services this means no algorithm can coordinate with all systems as proven in the Two Generals Problem. As databases become more and more distributed this coordination will increase the difficulty of making truly atomic commits.

Usage: Atomic commits are essential for multi-step updates to data. This can be clearly shown in a simple example of a money transfer between two checking accounts.

This example is complicated by a transaction to check the balance of account Y during a transaction for transferring 100 dollars from account X to Y. To start, first 100 dollars is removed from account X. Second, 100 dollars is added to account Y. If the entire operation is not completed as one atomic commit, then several problems could occur. If the system fails in the middle of the operation, after removing the money from X and before adding into Y, then 100 dollars has just disappeared. Another issue is if the balance of Y is checked before the 100 dollars is added, the wrong balance for Y will be reported.

With atomic commits neither of these cases can happen, in the first case of the system failure, the atomic commit would be rolled back and the money returned to X. In the second case, the request of the balance of Y cannot occur until the atomic commit is fully completed

In case of distributed transaction, the client has requested operations at more than one servers. Atomic commit protocols are designed to achieve this effect even if the server crashes during their execution. The two phase commit protocol allows a server to decide to abort unilaterally. It includes timeout actions to deal with delays due to the servers crashing. The two phase protocol can take unbounded amount of time to complete but is guaranteed to complete eventually.

Operations for two-phase commit protocol:

canCommit?(trans)-> Yes / No

- Call from the coordinator to participant to ask whether it can commit transaction. Participant replies with its vote.

doCommit(trans)

- Call from the coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

- Call from the coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)

- Call from the participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

- Call from the participant to coordinator to ask for the decision on a transaction after it has voted Yes but has still had no reply after some delay. Used to recover from server crash or delayed messages.

The two-phase commit protocol:

Phase 1:

The coordinator sendscanCommit? request to each of the participants in the transaction. When participant receives a canCommit? request it replies with its vote (Yes or No) to coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

Phase2:

The coordinator collects votes (including its own).

a) If there are no failures & all the votes are Yes the coordinator decides to commit the transaction and sendsdoCommit request to each of the participants.
b) Otherwise the coordinator decides to abort the transaction and sends doAbort requests to all participants that voted Yes.

Participants that voted Yes are waiting for a doCommit or doAbort request from the coordinator. When a participant receives one of these messages it acts accordingly & in the case of commit, makes a haveCommitted call as confirmation to coordinator.

Concurrency control in distributed transactions:

The concurrency control in distributed transaction:

1. **Locking**:

- The server attempts to lock any object that is about to use by client's transaction &Requests to lock objects are suspended and wait until the objects are unlocked.
- Serial equivalence: Serial equivalence requires that all of the transaction's accesses to a particular object be serialized with respect to accesses by other transactions. To achieve serial equivalence transaction is not allowed to any new locks after it has release a lock.

Two-phase lock: growing phase (new locks are acquired), shrinking phase (locks are released).

Strict execution: locks are held until transaction commits/aborts. (Strict two-phase locking).

Recoverability: locks must be held until all objects it updated have been written to permanent storage.

2. Timestamp ordering concurrency control:

Optimistic concurrency control is a concurrency control method applied to transactional systems such as relational database management systems & software transactional memory. Optimistic concurrency control assumes that multiple transactions can frequently complete without the interfering with each other. While running, transactions use data resources without acquiring locks on those resources. Before committing, each transaction verifies that no other transaction has modified the data it has read. If the check reveals conflicting modifications, committing transaction rolls back & can be restarted. In most applications, the likelihood of two client's transactions accessing the same object is low. Transactions are allowed to proceed as though there were no possibility of conflict with other transactions until client completes its task and issues a close Transaction request.

3-phases of a transaction:

- Working phase: each transaction has tentative version of each of the objects that it updates.
- Validation phase: when the closeTransaction request is received, transaction is validated to establish whether or not its operations on objects conflict with other transaction.
- Update phase: changes in tentative versions are made permanent if transaction is validated.

3. Optimistic concurrency control:

Each transaction is assigned a unique timestamp values when it starts. Timestamp defines its position in time sequence of transaction. The most commonly used concurrency protocol is timestamp based protocol. Protocol uses either system time or logical counter as a timestamp. Every data item is given latest read and write-timestamp. This lets the system know when last 'read and write' operation was performed on data item. Timestamp-ordering protocol ensures serializability among transactions in their conflicting read & writes operations. This is responsibility of protocol system that the conflicting pair of tasks should be executed according to timestamp values of the transactions.

Distributed deadlocks:

A transaction is deadlocked if it is blocked and will remain blocked until there is intervention. Locking-based CC algorithms may cause deadlocks. TO-based algorithms that involve waiting may cause deadlocks.

- Wait-for graph:

If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG. Shown in fig 4.4

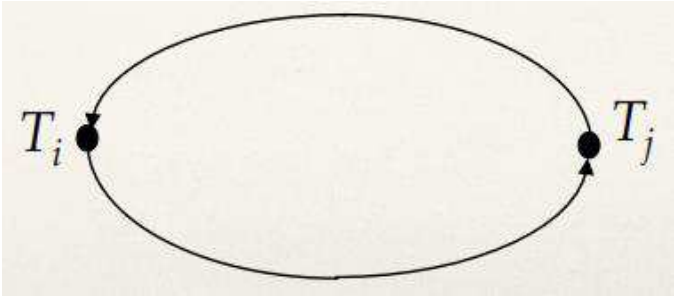

Fig 4.4 WFG for transection

All other theory and techniques are same as previous discussion.

Transaction recovery:

Customization of transaction recovery consists of:

- Determining which application program (TP) transactions will have recovery automation.
- Identifying the batch message region (BMP) transactions that will have recovery automation
- Specifying the error threshold level that a recovery should stop at
- Identifying specific abend codes that you want recovery procedures to occur for
- Specifying the recovery procedure, which usually consists of invoking a command, a routine, and/or sending notifications to an operator

The recovery itself is typically triggered from the NetView automation table by calling the EVIECT0X routine when certain messages arrive at NetView. EVIECT0X then consults the automation policy to find out whether recovery is to be attempted and to determine what has to be done.

Replication

Replication is a technique for enhancing a service. The motivations for replication are to improve a service is performance, to increase the availability or to it fault tolerant.

Two replication strategies have been used in distributed systems: Active and Passive replication. shown in fig .4.5.

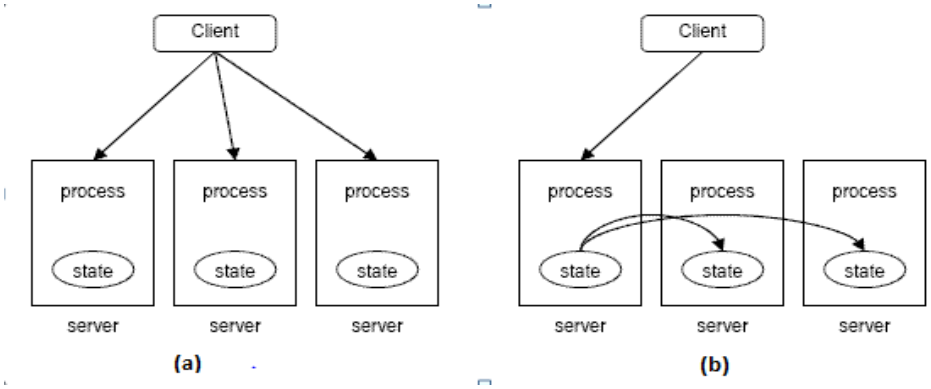

Fig 4.5 (a) Active replication (b) Passive replication

Active replication: In active replication each client request is processed by all the servers. Active Replication was first introduced by Leslie Lamport under the name state machine replication. This requires that the process hosted by the servers is deterministic. Deterministic means that, given the same initial state and a request sequence, all processes will produce the same response sequence and end up in the same final state. In order to make all the servers receive the same sequence of operations, an atomic broadcast protocol must be used. An atomic broadcast protocol guarantees that either all the servers receive a message or none, plus that they all receive messages in the same order. The big disadvantage for active replication is that in practice most of the real world servers are non-deterministic. Still active replication is the preferable choice when dealing with real time systems that require quick response even under the presence of faults or with systems that must handle byzantine faults.

Passive replication: In passive replication there is only one server (called primary) that processes client requests. After processing a request, the primary server updates the state on the other (backup) servers and sends back the response to the client. If the primary server fails, one of the backup servers takes its place. Passive replication may be used even for non-deterministic processes. The disadvantage of passive replication compared to active is that in case of failure the response is delayed.

System model and group communication:

System Model:

The data consists of collection of terms that called object. But each such logical object is implemented by a collection of physical copies called replicas. The system model provides for managing replicas and describes group communication.

The model involves replicas cells by distinct replicas managers which are the components that contain the replicas on a given computer and performed operation upon them directly.

Group Communication:

It occurs when one source process sending a message to a group of process. Destination is group rather than a single process**.**

1. **Broadcast:-**Destination is everybody.
2. **Multicast:-**Destination is a design group**.**
3. **Unicast:-**Destination is a single process**.**

Group Management Functions:-

1. **Forming a group:-**Create-group (out gid: group id)
   Return a group identifier.
2. **Joining a group:-**Join group(in gid: group id)
   Mix the color process member of a group id.
3. **Leaving a group:-**Leave group (in gid: group id)
   Remove the caller process from the group
4. **Delete a group:-**Delete group(in gid: group id)
   Only authorized process can delete a group

Fault-tolerant services:

- Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components.
- If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively designed system in which even a small failure can cause total breakdown.
- Fault tolerance is particularly sought after in high-availability or life-critical systems. The ability of maintaining functionality when portions of a system break down is referred to as graceful degradation.
- A fault-tolerant design enables a system to continue its intended operation, possibly at a reduced level, rather than failing completely, when some part of the system fails.
- Within the scope of an individual system, fault tolerance can be achieved by anticipating exceptional conditions and building the system to cope with them, and, in general, aiming for self-stabilization so that the system converges towards an error-free state. However, if the consequences of a system failure are catastrophic, or the cost of making it sufficiently reliable is very high, a better solution may be to use some form of duplication. In any case, if the consequence of a system failure is so catastrophic, the system must be able to use reversion to fall back to a safe mode. This is similar to roll-back recovery but can be a human action if humans are present in the loop.

Transactions with replicated data:

Transaction replication system, the Replication Agent and Replication Server components both provide features that allow you to identify the transactions that you want to replicate. You do not need to replicate all transactions, or all data-changing operations, in primary database. The ability to select transactions for replication is particularly useful when you need to implement replication system to support an application that uses some of tables in a database, but not all of them. By marking tables, you identify the specific tables in primary database for which transactions are replicated. Transactions that affect data in marked tables are referred to as replicated transactions.

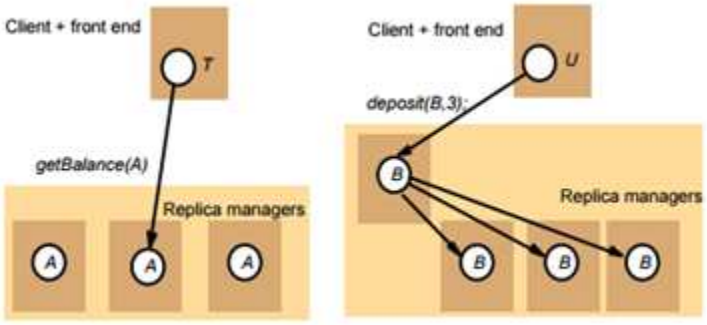

Fig 4.6. Transactions on replicated data

Note: If transaction affects data in both marked and unmarked tables, only operations that affect data in marked tables are replicated.

By marking stored procedures, you identify the specific procedures in the primary database that are to be replicated as applied functions. When a marked procedure is invoked in primary database, its invocation is replicated, along with its input parameter values, to the replicate database. The ability to select procedures for replication is particularly useful when you need to implement replication system to support an application that uses stored procedures, or when replicating a single procedure invocation is more efficient than replicating numerous, individual data-changing operations are produced by single procedure invocation. Shown in fig 4.6

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in