UNIT-5

1. A FRAMEWORK FOR TRANSACTION MANAGEMENT

We define the properties of transactions, goals of distributed transaction management, and model of distributed transaction in the framework of transaction management.

1. 1 Properties of transactions

A transaction is an application or part of the application which is characterized by following properties.

1.1.1. Atomicity

Either all or none of the transactions operations are performed. If any transaction is interrupted by a failure, then its partial results are undone

1.1.2. Transaction abort or system crashes

The abort of transaction can be requested by the transaction itself, because some of its inputs are wrong, and some conditions may indicate that the transaction is inappropriate or useless.

A transaction abort is performed by the system when it has overload and deadlocks of operations.

1.1.3. Transaction recovery

The activity of ensuring atomicity in the presence of transaction aborts is called transaction recovery.

The activity of ensuring atomicity in the presence of system crashes is called crash recovery.

1.1.4. Commitment

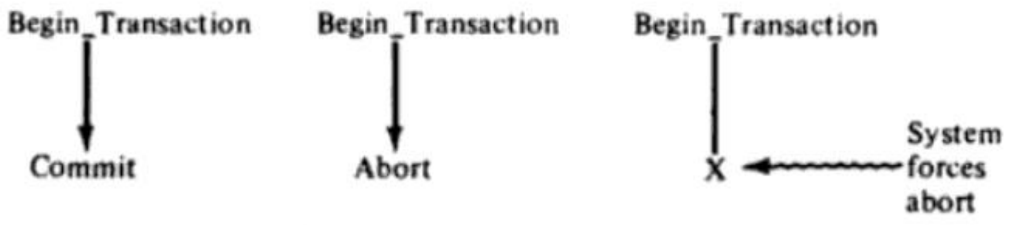
The completion of a transaction is called commitment, in which two primitives were identified, they are begin_transaction primitive and commit or abort primitive.

Each transaction begins with a begin_transaction primitive and ends with a commit primitive or an abort primitive.

1.1.5. Durability

Once a transaction has committed, the system must be guarantee that the results of the operations will never be lost and independent of subsequent failures.

Activity of providing the transactions durability is called database recovery.



Types of transaction termination.

1.1.6. Serializability and Concurrency control

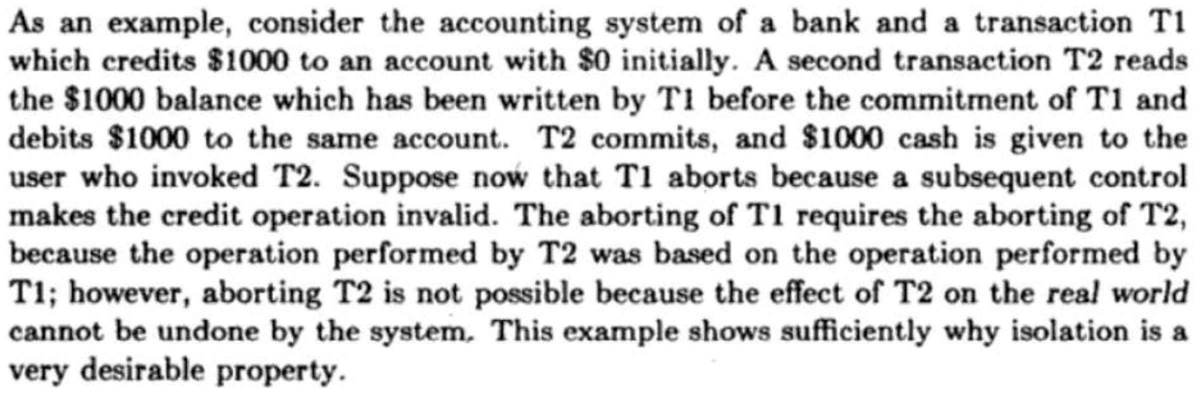
Several transactions are executed concurrently, the result must be the same as if they were executed in some other serial order.

The activity of guaranteeing transactions serializability is called concurrency control.

1.1.7. Isolation

An incomplete transaction cannot reveal its results to other transactions before its commitment and this kind of property is known as transaction isolation. This property is needed in order to avoid the problem of cascading aborts.

Example of Isolation:



As an example, consider the accounting system of a bank and a transaction T1 which credits \$1000 to an account with \$0 initially. A second transaction T2 reads the \$1000 balance which has been written by T1 before the commitment of T1 and debits \$1000 to the same account. T2 commits, and \$1000 cash is given to the user who invoked T2. Suppose now that T1 aborts because a subsequent control makes the credit operation invalid. The aborting of T1 requires the aborting of T2, because the operation performed by T2 was based on the operation performed by T1; however, aborting T2 is not possible because the effect of T2 on the *real world* cannot be undone by the system. This example shows sufficiently why isolation is a very desirable property.

1.1.2 Goals of Transaction Management

<u>Goals of transaction management are defined as follows:</u>

1. Efficient
2. Reliable
3. Concurrent execution of transactions

For achieving of above goals, the transaction management is mainly focused on following aspects:

a. **CPU and main memory utilization**
   This aspect is common to both centralized and distributed databases, although typical database applications are I/O bound i.e., they spent much time for I/O operation even the operations are small. In concurrent executions, special techniques are implement to overcome the problem of revealed bottleneck in main memory or in CPU time.
b. **Control messages**
   In distributed databases, another important aspect is efficiency; a control message is needed to control the execution of message and it is dependent on previous executions of processes. It is required to reduce the cost due to occurrence of number of control messages for improving the efficient when to run the applications in a distributed database.
c. **Response time**
   As the third important efficiency aspect, we consider the response time of each individual transaction. Response time is more critical for distributed applications than for local applications because of additional time is required for communication between different sites.

d. **Availability**

Availability of the whole system is considered as an another aspect in distributed system. Transaction manager must not block the failure transactions or non-operational transactions for execution.

Summarized goals of transaction management in a distributed database for the controlling of execution of transactions are as follows:

1. Transactions have atomicity, durability, serializability, and isolation properties
2. Their cost in terms of main memory, CPU, and number of transmitted control messages, and their response time are minimized
3. The availability of system is maximized

1.1.3 Distributed Transactions

A transaction is always part of an application. When a user types an application code, and request the application for execution, which does not have the properties of atomicity, durability, serializability, and isolation. After starting of application by user request, the application issues a begin transaction primitive, and all actions performed by the application, until a commit or abort primitive is issued.

In order to perform functions at different sites, a distributed application has to execute several processes at these sites. We will call these processes the agents of the application. An agent is therefore a local process which performs some actions on behalf of an application.

In order to cooperate in the execution of the global operation required by the application, the agents have to communicate. As they are residing at different sites, the communication between agents is performed through messages.

For achieving the atomicity, the following steps are to be performed in distributed databases.

1. There exists a root agent which starts the whole transaction, so that user requests the execution of an application, the root agent is started; the site of the root agent is called the **site of origin** of the transaction.
2. The root agent has the responsibility of issuing the begin_transcation, commit and abort primitives
3. Only the root agent can request the creation of a new agent.

While assumption 1 is intrinsic to the definition of root agent, the other two assumptions are made only for simplifying the presentation, in which, assumption 2 is adopted by most systems, only few systems adopt assumption 3.

```
FUND_TRANSFER:
Read(terminal, $AMOUNT, $FROM_ACC, $TO_ACC);
Begin_transaction;
Select AMOUNT into $FROM_AMOUNT
from ACCOUNT
where ACCOUNT_NUMBER=$FROM_ACC;
if $FROM_AMOUNT - $AMOUNT < 0 then abort
else begin
            Update ACCOUNT
            set AMOUNT=AMOUNT - $AMOUNT
            where ACCOUNT=$FROM_ACC;
            Update ACCOUNT
            set AMOUNT=AMOUNT + $AMOUNT
            where ACCOUNT=$TO_ACC;
            Commit
         end
```

(a) The FUND_TRANSFER transaction at the global level

```
ROOT-AGENT:
Read(terminal,$AMOUNT, $FROM_ACC, $TO_ACC);
Begin_transaction;
Select AMOUNT into $FROM_$AMOUNT
from ACCOUNT
where ACCOUNT_NUMBER=$FROM_ACCOUNT;
If $FROM_AMOUNT - $AMOUNT < 0 then abort
else begin
            Update ACCOUNT
            set AMOUNT=AMOUNT - $AMOUNT
            where ACCOUNT=$FROM_ACC;
            Create AGENT1;
            Send to AGENT1($AMOUNT, $TO_ACC);
            Commit
         end

AGENT1:
Receive from ROOT_AGENT ($AMOUNT, $TO_ACC);
Update ACCOUNT
set AMOUNT=AMOUNT + $AMOUNT
where ACCOUNT=$TO_ACC;
```

(b) The FUND_TRANSFER transaction constituted by two agents

FUND_TRANSFER application.

2. SUPPORTING ATOMICITY OF DISTRIBUTED TRANSACTIONS

The global primitives begin_transaction, commit, and abort must be implemented by executing a set of appropriate local actions at the sites where the distributed transaction is executed. Each site has **local transaction manager (LTM)**, which is capable of implementing local transactions.

2.1 Recovery in centralized systems

Recovery mechanism is designed for allowing the normal operation of a database after a failure. In the design of effective recovery, we analyze the following aspects.

2.1.1 A model of failures in centralized databases

From the viewpoint of local recovery, we consider the most important characteristic of failure is the **amount of information which is lost because of the failure**. Key classification of failures are described as follows:

a. Failure without loss of information : these failure include, abort transaction, error condition, arithmetic overflow (like division by zero). In these failures, all the information is stored in memory is available for the recovery.
b. Failures with loss of volatile storage: In these failures, content of main memory is lost; however, the information which is recoded on disk is not affected by the failure. System crashes is an example of this failure
c. Failures with loss of nonvolatile storage: In these failures, called **media failures**, the content of disk storage is lost. Head crashes in the disk is an example of this kind of failure.
d. Failures with loss of stable storage: Some information store in stable storage is lost because of several simultaneous failures of the third type.

2.1.2 Logs

The basic technique for implementing transactions in presence of failures is based on the use of logs. A log contains information for undoing or redoing of all actions which are performed by transactions.

Undo the actions of transaction means to reconstruct the database as prior to its execution. If the commitment of transaction is not possible due to the presence of failure, then it is required to form the database as remains same of prior to start the transaction

Redo the actions of transaction means to perform again its actions.

Undo and Redo are idempotent i.e., UNDO(UNDO(UNDO(…(action)….))) = UNDO(action); REDO(REDO(UNDO(…(action)….))) = REDO(action).

The log record contains the following information

1. Identifier of the transaction
2. The identifier of the record
3. The type of action (insert, delete, modify)
4. The old record value (required for the undo)

5. The new record value (require for the redo)
6. Auxiliary information for the recovery procedure (a pointer to the previous log record of the same transaction)

2.1.3 Recovery procedures

When a failure with loss of volatile storage occurs, a recovery procedure reads the log file and performs the following operations

a. Determine all non-committed transactions that have to be undone. Non-committed transactions are recognized because they have a begin_transaction record in the log file without having a commit or abort the record
b. Determine all transactions which need to be redone. In principle, this set includes all transactions a commit record in the log file. Most of transactions are safely stored into in stable storage before the failure and they do not need to redone operation. Some of transactions need to redone with **checkpoints.**
c. 'Step a' shows the determining of undo transactions; 'Step b' shows the redo determined transactions.
   "**Checkpoints** are operations which are periodically performed". Performing of checkpoints requires the following operations.

   i. Writing to stable storage all log records and all database updates which are still in volatile storage
   ii. Writing to stable a check point is record. Checkpoint indicates that which transactions are active at that particular time.

   <u>The existence of checkpoints facilitates the recovery procedure. Refinements of recovery procedure is defined as follows:</u>

   1. Find and read the last check point record
   2. Put all transactions written in the checkpoint record into the undo set, which contains the transactions to be undone. The redo set is initially empty
   3. Read the log file starting from the checkpoint record until its end. If a begin_transaction is found, put the corresponding transaction in the undo set. If a commit record is found, move the corresponding transaction from the undo set to redo set.

2.2 Communication Failures in Distributed Databases

We have examined the failures which can occur at each site called **site failure**. Recovery mechanisms for distributed transactions require at failures in the communication between sites.

<u>When a message is sent from site X to site Y, we require the communication behavior as per follows:</u>

i. X receives positive acknowledge after a delay which is less than some maximum delay DMAX
ii. The message is delivered at Y in proper sequence with respect to other X-Y messages
iii. The message is correct

There is a great chance to occur failures with respect to above specification; for example, the message might not be correct or the message be out of order, X is not received the acknowledgement with the message being delivered, X is received the acknowledgement without the message being delivered. In such cases, we assume the following steps for ignoring the errors. If a message from X to Y is delivered at Y, then message is correct and is in sequence with respect to other X-Y messages

If X receives an acknowledgement, then the message has been delivered

2.3 Recovery of distributed transactions

Each agent issues the following primitives:

begin_transaction, commit, and abort primitives

After issuing begin_transaction to its LTM, an agent will possess the properties of local transaction and calling of agent which issues a begin_transaction primitive to its LTM as 'a subtransaction'.

Local primitives issued by each agent to its LTM called the following primitives:

local_begin, local_commit, and local_abort.

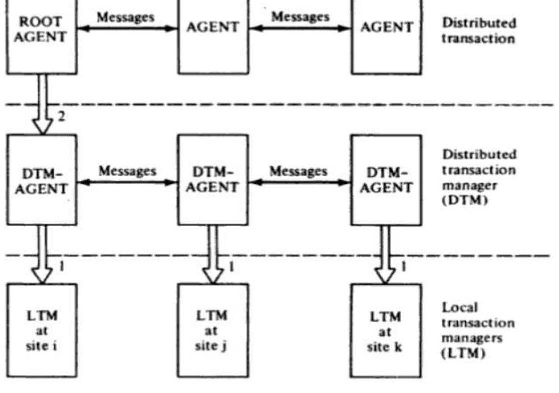
Building a distributed transaction manager (DTM) has require the following properties:

1. Ensuring the atomicity of a subtransaction
2. Writing some records on stable storage on behalf of the distributed transaction manager

For the first property of DTM, it is necessary to apply the following conditions

1. At each either all actions are performed or none of the operations are performed
2. All sites must take the same decision with respect to the commitment or abort of subtransactions

The relationship between distributed transaction management and local transaction management is represented in the reference model of a distributed transaction manager as shown in following figure.



Interface 1: Local_begin, Local_Commit, Local_Abort, Local_Create
Interface 2: Begin_Transaction, Commit, Abort, Create

Figure A reference model of distributed transaction recovery.

At the bottom level, we have local transaction managers and communication between them, in which the LTMs implement the interface 1 of “local_begin, local_commit, local_abort, and local create”. Local_create is used for creation of a process (agent) is a function of the local system.
At the next higher level, the DTM will be implemented by a set of local DTM-agents which exchange the messages between them. DTM implements the interface 2 of “begin_transaction, commit, and abort primitives” . Interface 2 is used only by the root agent.

Begin_transaction

When a begin_transaction is issued by the root agent, the DTM will have to issue a local_begin primitive to the LTM at the site of the origin, in which all agents of the sites are transformed into subtransactions.

Example of this process is illustrated in following figure for the FUND_TRANSFER application.

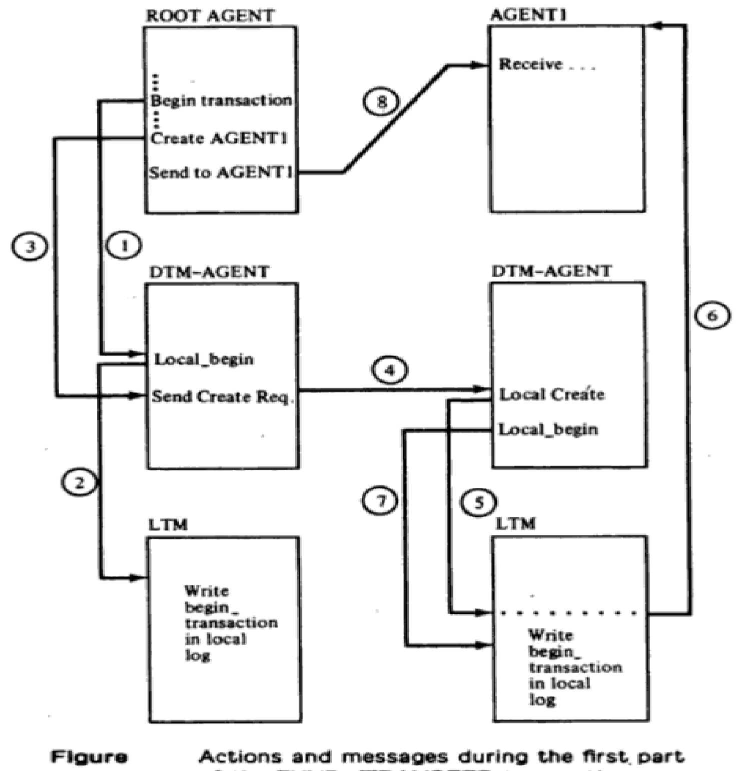


Figure Actions and messages during the first part of the FUND_TRANSFER transaction.

Abort:

When an abort issued by the root agent, all existing subtransactions must be aborted. This is performed by issuing local_aborts to the LTMs at all sites where there is active subtransaction. All subtransactions are aborted then it obtains the global_abort

Commit:

It is expensive and more difficult task. It requires that all subtransactions must be commit even in the case of failures. However, it is not acceptable that one of subtransactions is locally aborted because of the failure and the other subtransactions are having with commit. In such cases, 2-phase commitment (2PC) is developed for ensuring either global_commit or global_abort.

2.4 The 2-Phase Commitment Protocol (2PC)

In 2-phase commitment protocol, DTM-agent performs as a special role called “**coordinator**”, all other agents which must commit together are called **participants.**

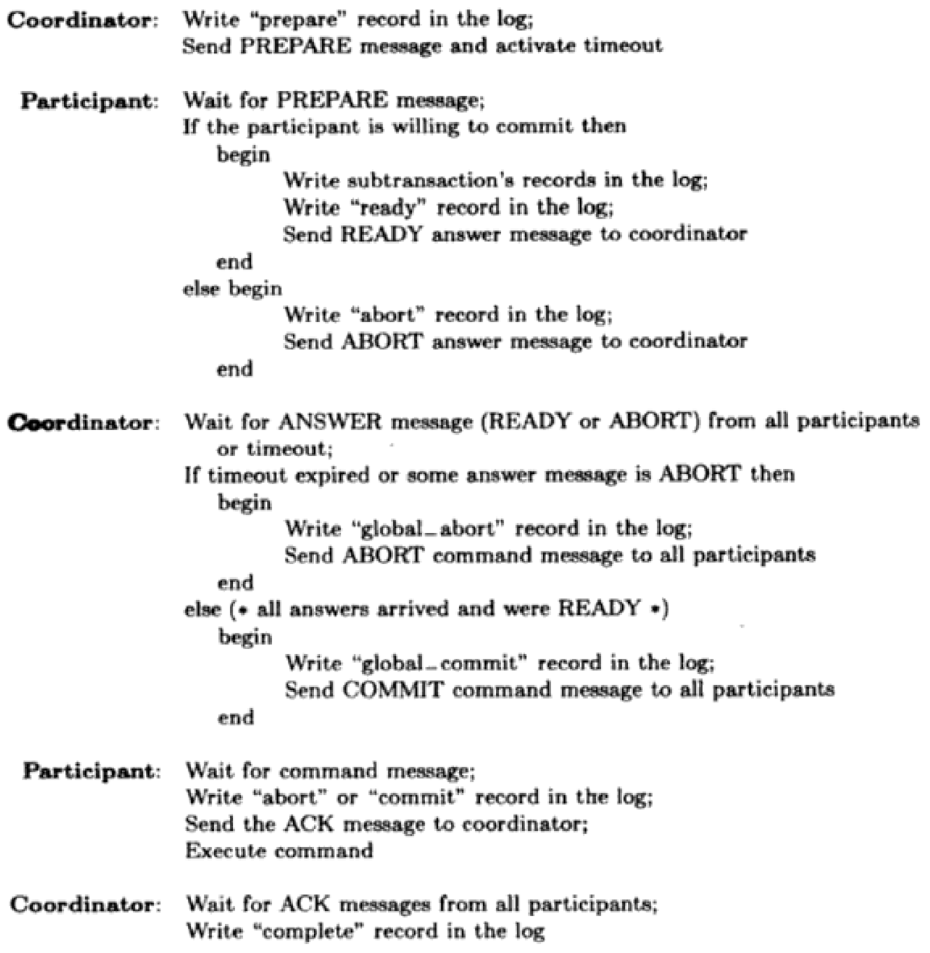
The coordinator is responsible for taking the final commit or abort decision.

Each participant corresponds to a subtransaction which performs write action and we assume that each participant is at different site.

The basic idea of 2PC is as follows:

“It determine a unique decision for all the participants with respect to committing or aborting all the local transactions. If the participant is unable to perform commit, then all the participants must locally abort.”

The illustrated procedure of 2PC is shown in below figure.



```
Coordinator:  Write "prepare" record in the log;
              Send PREPARE message and activate timeout

 Participant: Wait for PREPARE message;
              If the participant is willing to commit then
                  begin
                          Write subtransaction's records in the log;
                          Write "ready" record in the log;
                          Send READY answer message to coordinator
                  end
              else begin
                          Write "abort" record in the log;
                          Send ABORT answer message to coordinator
                  end

Coordinator:  Wait for ANSWER message (READY or ABORT) from all participants
                  or timeout;
              If timeout expired or some answer message is ABORT then
                  begin
                          Write "global_abort" record in the log;
                          Send ABORT command message to all participants
                  end
              else (* all answers arrived and were READY *)
                  begin
                          Write "global_commit" record in the log;
                          Send COMMIT command message to all participants
                  end

 Participant: Wait for command message;
              Write "abort" or "commit" record in the log;
              Send the ACK message to coordinator;
              Execute command

Coordinator:  Wait for ACK messages from all participants;
              Write "complete" record in the log
```

Figure Basic 2-phase-commitment protocol.

The 2 PC describes the following two phases:

Phase one:

Goal of phase one is to reach the common decision.

The coordinator initially records log on stable storage, in which identifiers of subtransactions participating to the 2-phase commitment are recorded and the coordinator activates the time out mechanism.

When a participant answers READY, it ensures that it able to commit even failure is occurred at respective site.

The coordinator asks all the participants to prepare for commitment; each participant answers READY if it is ready to commit and willing to do for the same.

If all the participants have answered READY, it decides to commit the transaction. If any one of participants has answered ABORT or time out is expired, then coordinator taking the global decision as ABORT.

Phase two:

In phase two, the coordinator is able to implement the common decision based on received messaged of participants, in which the coordinator either to write a "global_commit" or "global_abort" record in the log. The coordinator informs all participants of its decision by sending the command message.

All participants send the final acknowledgement (ACK) message to the coordinator; when the coordinator has received the an ACK message from all its participants, then the coordinator write to implement as "complete record".

The behavior of 2PC is analyzed in the presence of failures as per follows:

a. **site failures**
    i. A participant failed before having written the ready record in the log
    In this case, the coordinator time out is expired, and it takes the abort decision.
    ii. A participant failed after having written the ready record in the log
    In this case, the operational sites correctly terminate the transaction (either commit or abort). When the failed site recovers, the restart procedure has to ask the coordinator or some other participant about the outcome of the transaction, and then perform appropriate action.
    iii. The coordinator fails after having written the prepare record in the log, but before having a written a global_commit or global_abort in the log
    In this case, all participants which have already answered READY must wait for the recovery of the coordinator.
    iv. The coordinator fails after having written a global_commit or global_abort record in the log, but before having the complete record written in the log
    In this case, the coordinator at restart must send to all participants the decision again; all participants which have not received the command have to wait until the coordinator recovers.
    v. The coordinator fails after having written the complete record in the log
    In this case, the transaction is already concluded, and no action is required at restart.
b. **lost messages**
    i. An answer message (READY or ABORT) from a participant is lost
    In this case the coordinators timeout expires, and the whole transaction is aborted.
    ii. A PREPARE message is lost
    In this case the participant remains in wait. The global result is same as in the previous

iii. A command message (COMMIT or ABORT) is lost
The destination participant remains uncertain about the decision.

iv. An ACK message is lost
The coordinator remains uncertain about the fact that the participant has received the command message. This problem can be eliminated by introducing a time out in the coordinator; if no ACK is received after the time out, the coordinator will send the command again.

c. Network Partitions
Dividing the sites in two groups; the group which contains the coordinator is called the **coordinator group**; the other **participant group.** From the viewpoint of the coordinator, the partition is equivalent to a multiple failure of a set of participants and a situation is similar to a-i and a-ii. From the view point of participant group, the partition is equivalent to a coordinator failure and a situation is similar to a-iii and a-iv.

Some Comments on the 2-Phase Commitment Protocol:

- **Unilateral abort capability:** Each site is authorized to unilaterally abort its sub-transaction until it has answered READY to the prepare message. However, after a sub-transaction has entered the ready state, this type of site autonomy is lost.
- **Blocking:** A problem with the basic 2-phase-commitment protocol is that a sub-transaction which has entered its ready state could be blocked.
- **Elimination of the Prepare Message:** When agents finish to perform their operations, they can return the READY message immediately, without waiting for a PREPARE message.

3. CONCURRENCY CONTROL FOR DISTRIBUTED TRANSACTIONS

We discuss the concurrency control in centralized and distributed databases, which are explained as follows:

3.1 Concurrency control based on locking in centralized databases

Basic Idea of Locking Problem:

“If a transaction wants to lock the data item which is already locked by another transaction, so that transaction must wait until other released the lock (unlock)”

Locking Modes

i. Shared Mode – a transaction locks the data item in shared mode if it wants to only read the data item.
ii. Exclusive Mode – a transaction locks the data item in exclusive mode if it wants to write the data item.

A transaction is well-formed

A transaction is well-formed if it always locks a data item in shared mode before reading it and it always locks a data item in exclusive mode before writing it.

Compatibility Rules Exist between Lock Modes

i. a transaction can lock a data item in shared mode if it is not locked at all or it is locked-in shared mode by another transaction
ii. a transaction can lock a data item in exclusive mode only if it is not locked at all

Thus, any two transactions are in conflict if they want to lock the same data item with two incompatible modes; i.e. shared-exlusive mode (or read-write) conflicts and exclusive-exclusive (or write-write) conflicts.

'Granularity of locking' refers to size of the objects are locked with a single lock operation. The following are different granularities of locking

i. it is possible lock at a 'record level' i.e. to lock individual tuples
ii. to lock at a 'file level' i.e. to lock entire fragments or relations

Concurrent execution of transactions is correct when the following rules are observed:

1. transaction abort
2. compatibility rules for locking are observed
3. each transaction does not request new locks after it has released a lock

Third rule is expressed the 2-phase-locking (2PL), in which new locks are acquired in growing phase, and locks are released in shrinking phase. This two phase locking is sufficient to achieve serializability.

2-phase-locking are well-formed and isolated when it follows the below schema

```
(Begin application)
Begin_transaction
Acquire locks before reading or writing
Commit
Release locks
(End Application)
```

Important problem is identified for locking mechanism is dead lock.
" a dead lock between two transactions arises if each transaction has locked a data item and is waiting to lock a different data item which has already locked by another transaction with a conflicting lock mode"

3.2 Concurrency control based on locking in distributed databases

Local transaction manager (LTM) allow a local agent to lock and unlock local data items. Reference model of distribute concurrency control is shown in below diagram.

If distributed transactions are well-formed and 2-phase locked (acquire locks in growing phase and release the locks in shrinking phase), then 2-phase locking is a correct mechanism in distributed databases as well as in centralized databases.

<u>The distributed transaction manages analyze two problems for solving.</u>

1. **Dealing with multiple copies of the data** – in redundancy of distributed databases, two transactions having conflicting locks on two copies of the same data item stored at different sites. In such cases, we issue a lock to all LTMs at all sites where a local copy of the data item is stored. Detailed analysis of this problem as follows:

a. Write-locks-all, read-locks-one : In this scheme, exclusive locks are acquired on all copies, while shared locks are acquired only on one arbitrary copy. A shared-exclusive conflict is detected at the site where the shared lock is required and exclusive conflicts are detected at all sites

b. Majority locking: Both shared and exclusive locks are requested at a majority copies of the data item

c. Primary copy locking: All locks must be required at this copy so that conflicts are discovered at the site where the primary copy resides

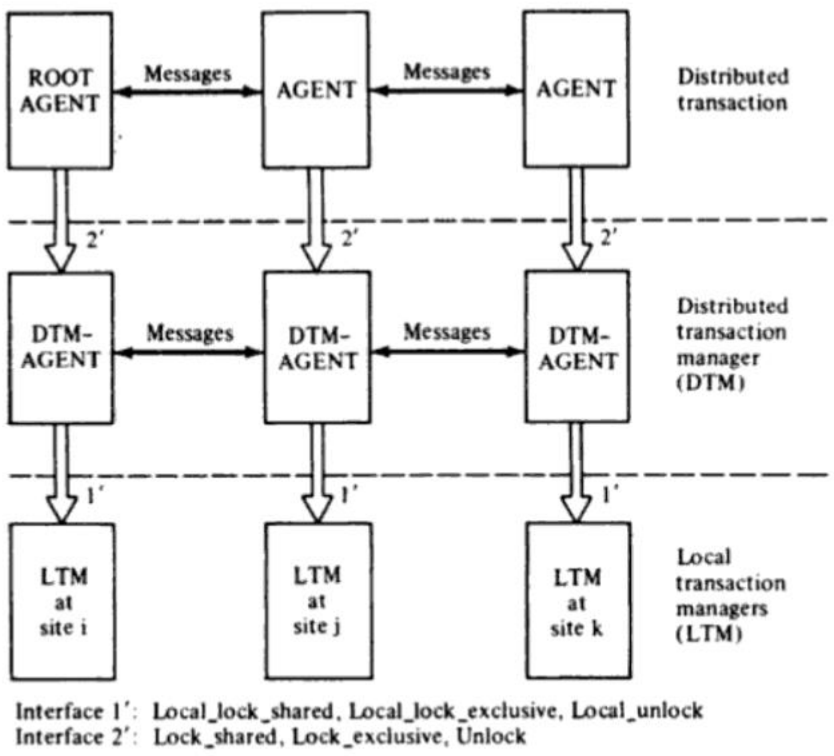

ROOT AGENT
Messages
AGENT
Messages
AGENT
Distributed transaction
2'
2'
2'
DTM-AGENT
Messages
DTM-AGENT
Messages
DTM-AGENT
Distributed transaction manager (DTM)
1'
1'
1'
LTM at site i
LTM at site j
LTM at site k
Local transaction managers (LTM)
Interface 1': Local_lock_shared, Local_lock_exclusive, Local_unlock
Interface 2': Lock_shared, Lock_exclusive, Unlock

Fig: A Reference Model for Distributed Concurrency Control.

2. **Deadlock detection**
   This is the second problem is faced by the DTM, in which a dead lock is a circular waiting situation which involve many transactions. Each transaction waits for another transaction and it can be represented with a **wait-for-graph.** A wait-for-graph is a directed graph having transactions as nodes, and edges from transaction T1 to T2 denotes that T1 waits for T2. If any cycle is occurred in wait-for-graph, then it said that dead lock is detected.

3.3 Comments on Distributed 2-Phase-Locking

Two-phase locking and availability: Locking to either all copies or majority of them, or a particular copy of data item can cause the execution of a transaction in case of failure.

Two-phase locking and recovery : all locking mechanisms sometimes require aborting a transaction because a deadlock is detected

4. ARCHITECTURAL ASPECTS OF DISTRIBUTED TRANSACTIONS

We consider the following are important aspect when we process the distributed transactions.

4.1 Processes and Servers

In a centralized database, we organize the transaction processing with single operating system for each (concurrent) transaction. There is strong correspondence between the entities of transaction manager (transactions) and entities of operating system (processes). This kind of organization is called as process model. In this switching time between main memory and I/O operation is very expensive. So that we use 'server model', in which transactions require services from the server process through request message.

4.2 Sessions and Datagrams

The communications between processes and servers can be performed through sessions and datagrams. The authentication and identification functions need to be performed only once and then messages can be exchanged without repeating these operations.

4.3 Computational structure of distributed transactions

The computation structure of distributed database transactions is organized in two ways: the centralized structure and the hierarchical structure.

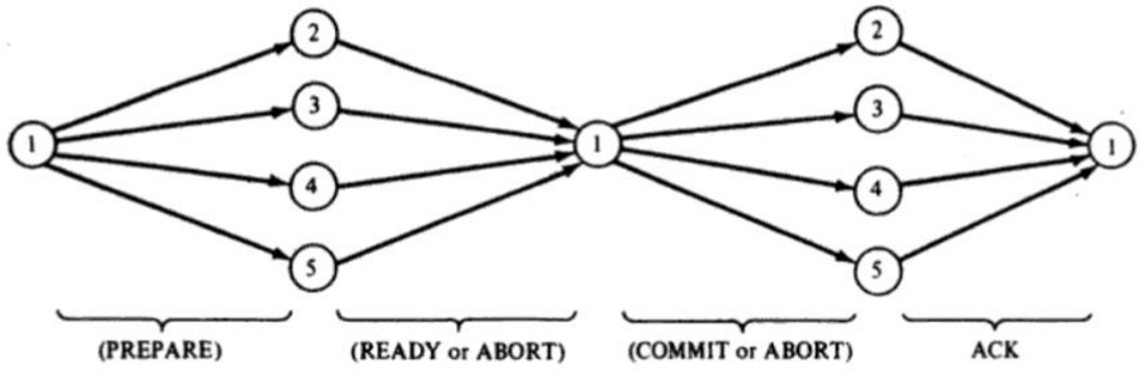
In the centralized structure, one agent (i.e. the root agent) activates and controls all other agents, in which other agents should not communicate between themselves.

In hierarchical structure, each agent can activate other agents, thus creating a tree of agents having the root agent as root. Direct communication between these models is intrinsic to this model.

4.4 Communication structure for commit protocols

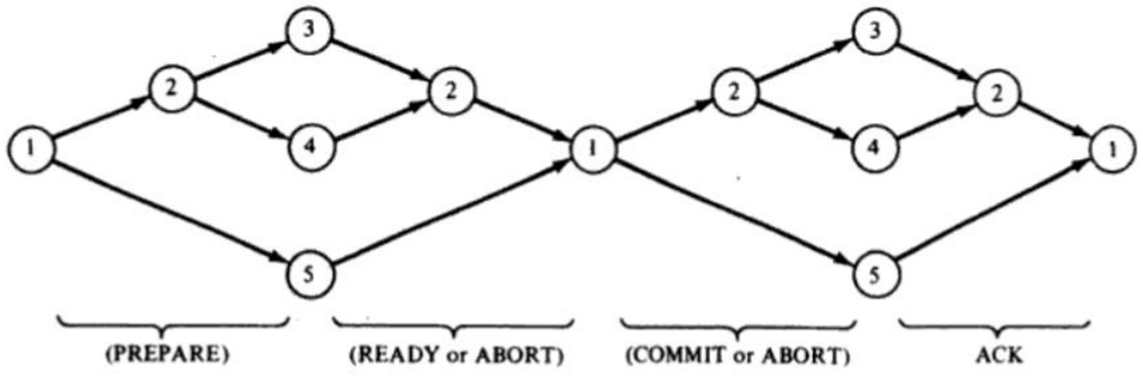
We analyze different communication structures for the DTM agents in order to implement commit protocol.

The 2PC commit protocol requires a centralized communication structure as per following figure. The communication is always performed between the coordinator DTM agent and the participants, but not between the participants directly.



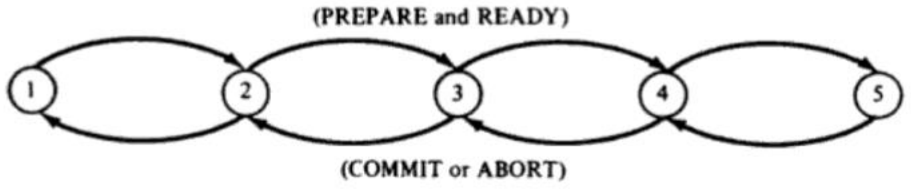
(a) Centralized.

Hierarchical communication protocol is shown in below, in which the coordinator is the DTM agent at the root of the tree. The communication between the coordinator and the participant is performed not by a direct broadcast, but by propagating the messages up and down of the tree.



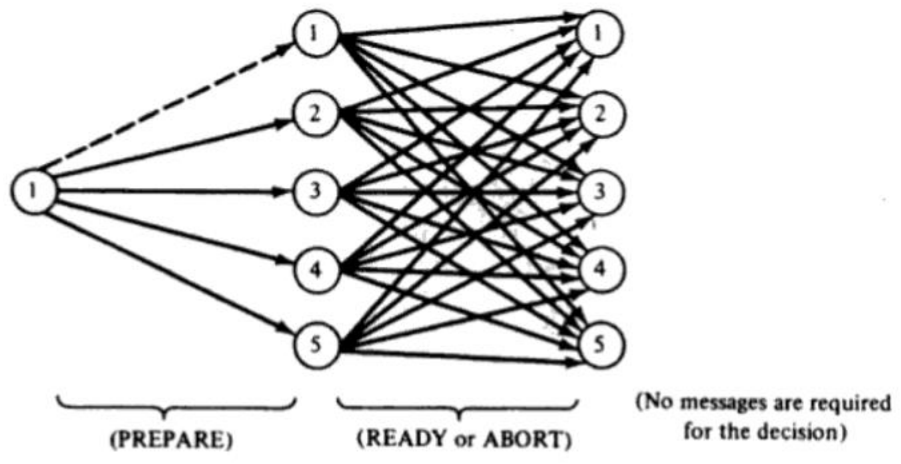
(*b*) Hierarchical.

Another class of communication protocol is the linear protocol, in which each site except the first and the last one has a predecessor and a successor. Instead of broadcasting a message from the coordinator to all the participants, the message is passed from each participant to its successor.



(*c*) Linear.

A different class of communication protocol is the distributed protocol, which is shown in below. A distributed protocol for commitment requires that each DTM -agent communicate with each other participant. The number of messages needed by a distributed protocol is much greater than the number of messages which is required by a centralized or hierarchical protocol.



(*d*) Distributed