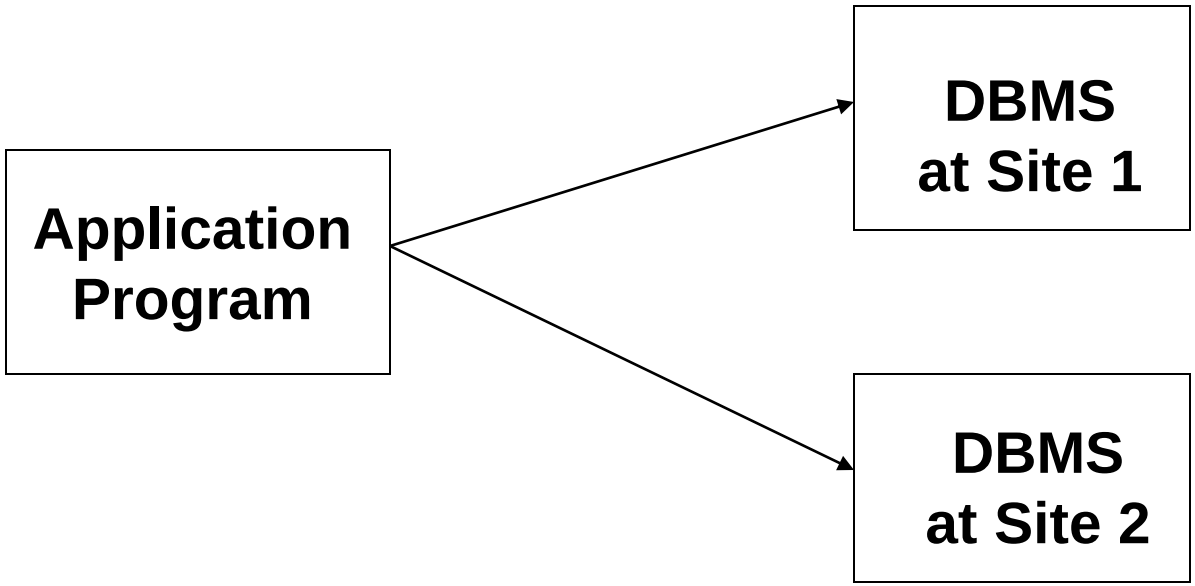
MANAGEMENT OF DISTRIBUTED TRANSACTIONS

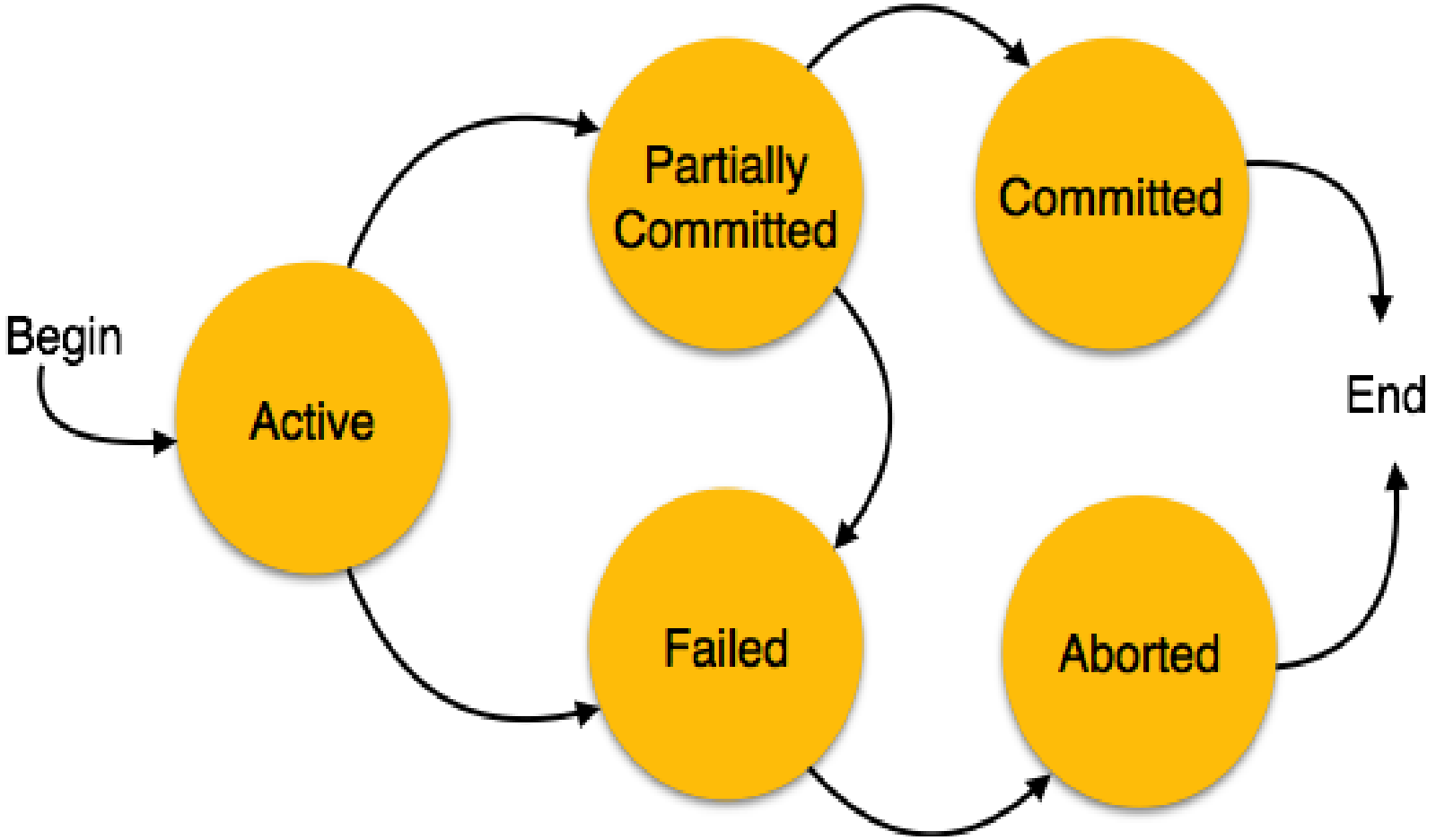
Unit III

Distributed Transaction

- **A distributed transaction accesses resource managers distributed across a network**



TRANSACTION



TRANSACTION

- A transaction is a program including a collection of database operations, executed as a logical unit of data processing.
- The operations performed in a transaction include one or more of database operations like insert, delete, update or retrieve data.

. read_item() – reads data item from storage to main memory.

. modify_item() – change value of item in the main memory.

. write_item() – write the modified value from main memory to storage.

Transaction Operations

- **The low level operations performed in a transaction are −**

1. **begin_transaction − A marker that specifies start of transaction execution.**
2. **read_item or write_item − Database operations that may be interleaved with main memory operations as a part of transaction.**
3. **end_transaction − A marker that specifies end of transaction.**
4. **commit − A signal to specify that the transaction has been successfully completed in its entirety and will not be undone.**
5. **rollback − A signal to specify that the transaction has been unsuccessful and so all temporary changes in the database are undone. *A committed transaction cannot be rolled back.***

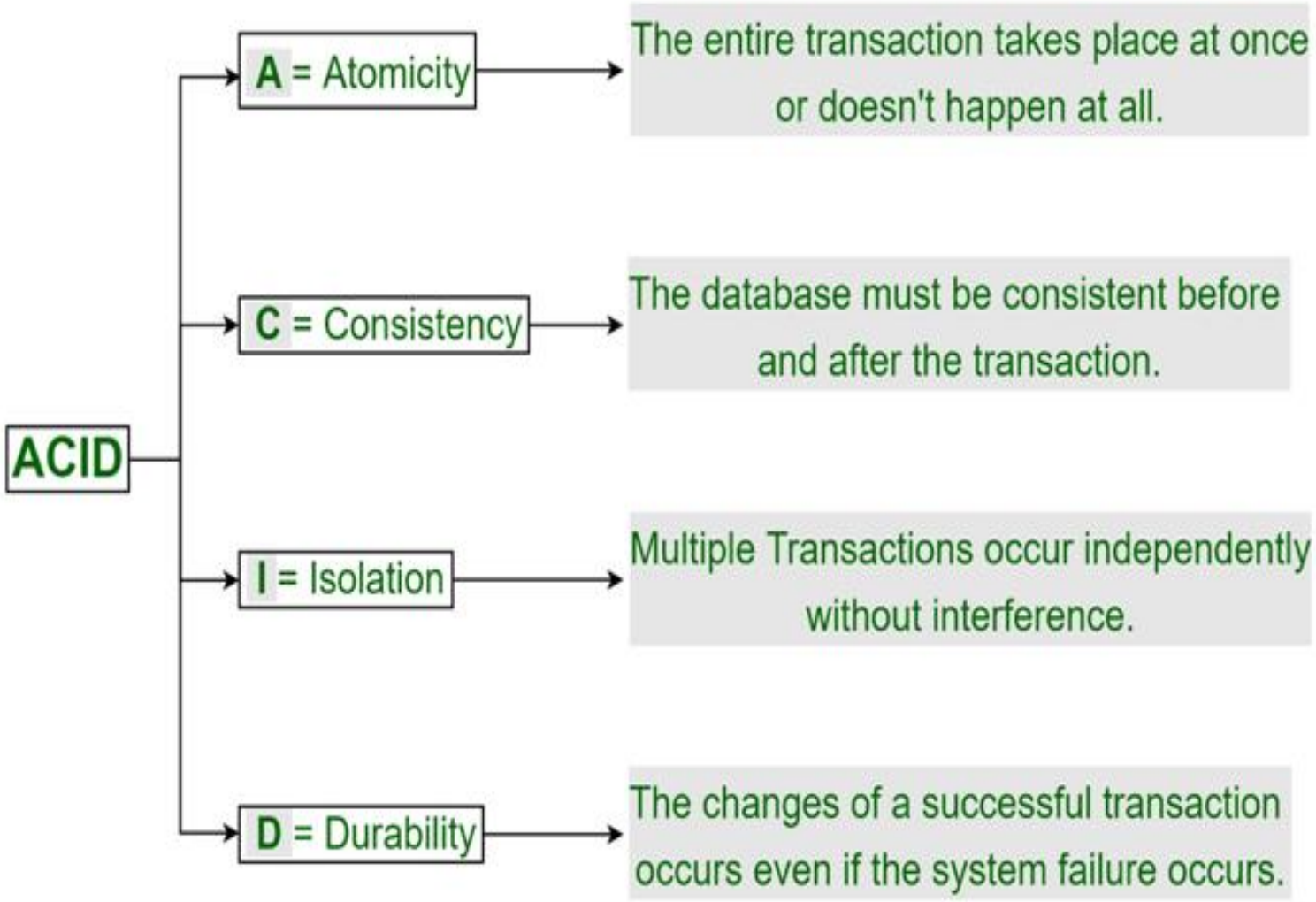
States of a transaction

- **Active:** Initial state and during the execution
- **Partially committed:** After the final statement has been executed
- **Committed:** After successful completion
- **Failed:** After the discovery that normal execution can no longer proceed
- **Aborted:** After the transaction has been rolled back and the DB restored to its state prior to the start of the transaction. Restart it again or kill it.

Goal of Transaction Management

1. Transactions have atomicity, durability, serializability and isolation properties.
2. CPU and main memory utilization
3. Control messages
4. Response time
5. Availability

ACID Properties in DBMS



Key points

Logs:

A log contains information for undoing or redoing all actions which are performed by transactions.

The log record contains

- **Identifier of the transaction**
- **Identifier of the record**
- **Need to be redone.**

Type of action(insert, delete, modify)

- **Old record value**
- **New record value**
- **Auxiliary information for the recovery procedure**

Recovery procedures

When a failure occurs a recovery procedure reads the log file and performs the following operations,

1.Determine all noncommitted transactions that have to be undone

2.Determine all transactions which need to be redone.

3.Undo the transactions determined at step 1 and redo the transactions determined at step 2.

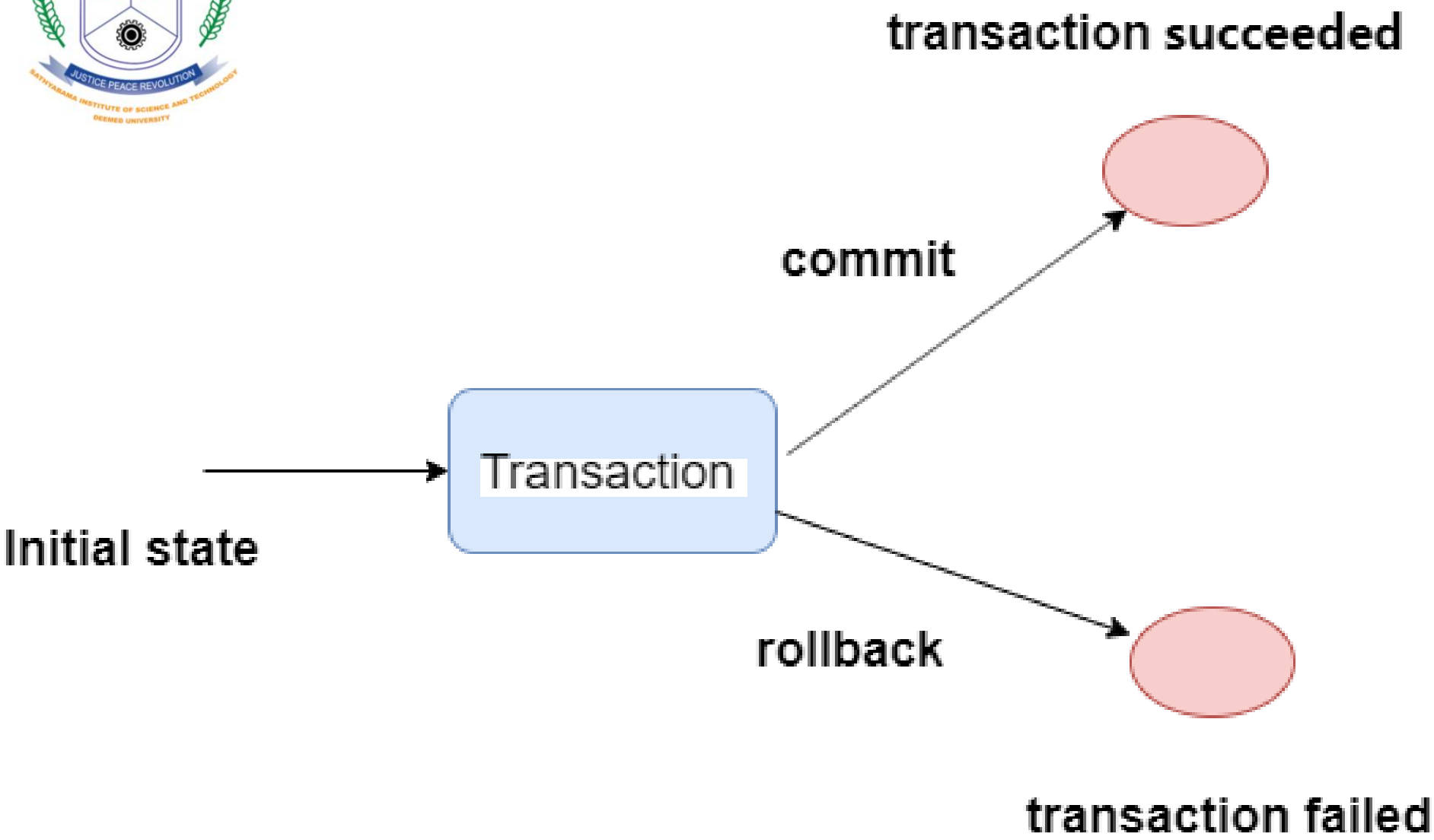
Distributed transaction management

- **Distributed transaction management deals with the problems of always providing a consistent distributed database in the presence of a large number of transactions (local and global) and failures (communication link and/or site failures).**
- **This is accomplished through**

  (i) distributed commit protocols that guarantee atomicity property;

  (ii) distributed concurrency control techniques to ensure consistency and isolation properties; and

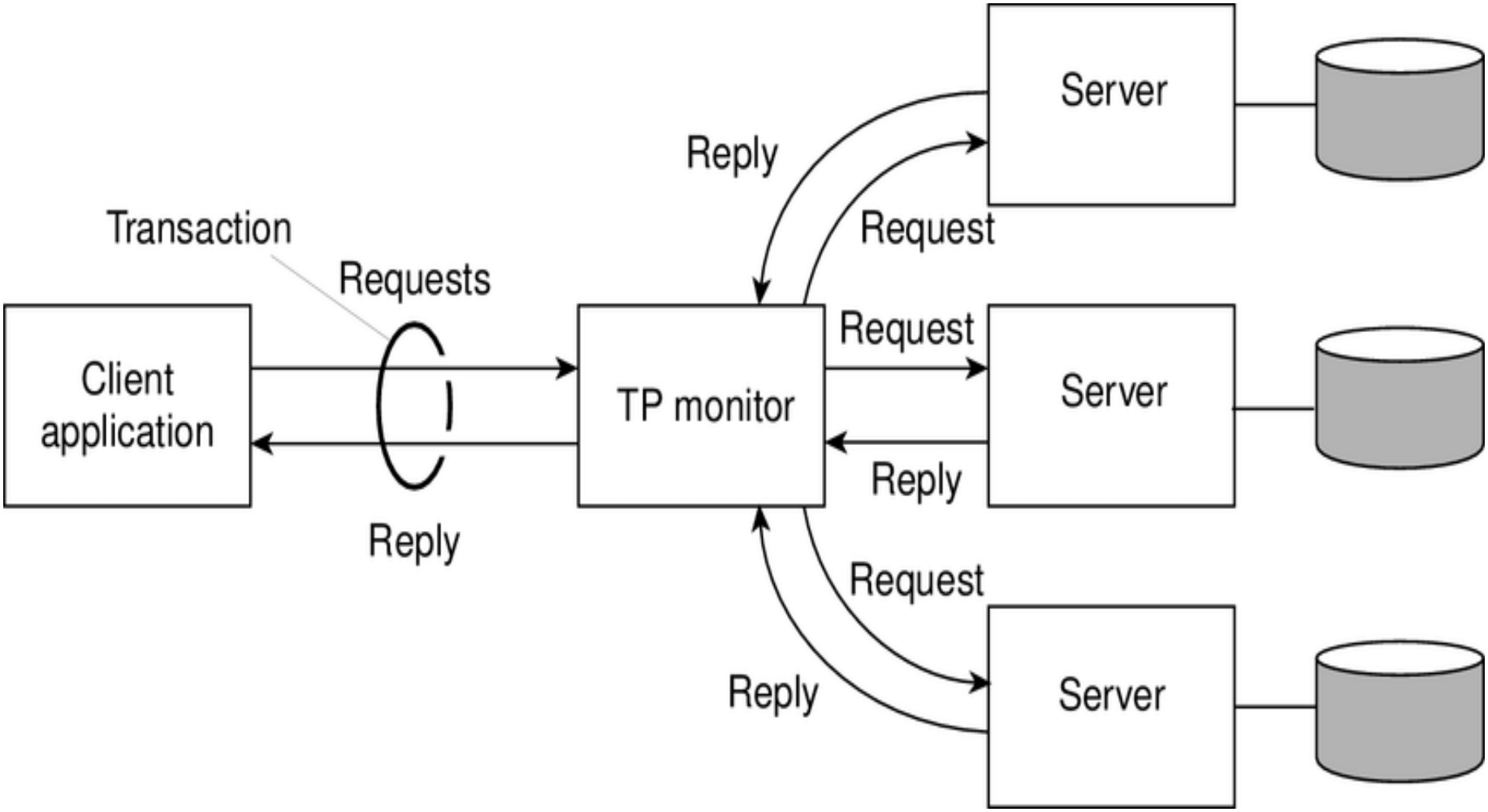
  (iii) distributed recovery methods to preserve consistency and durability when failures occur.


transaction succeeded
commit
Transaction
Initial state
rollback
transaction failed

Distributed Database Systems

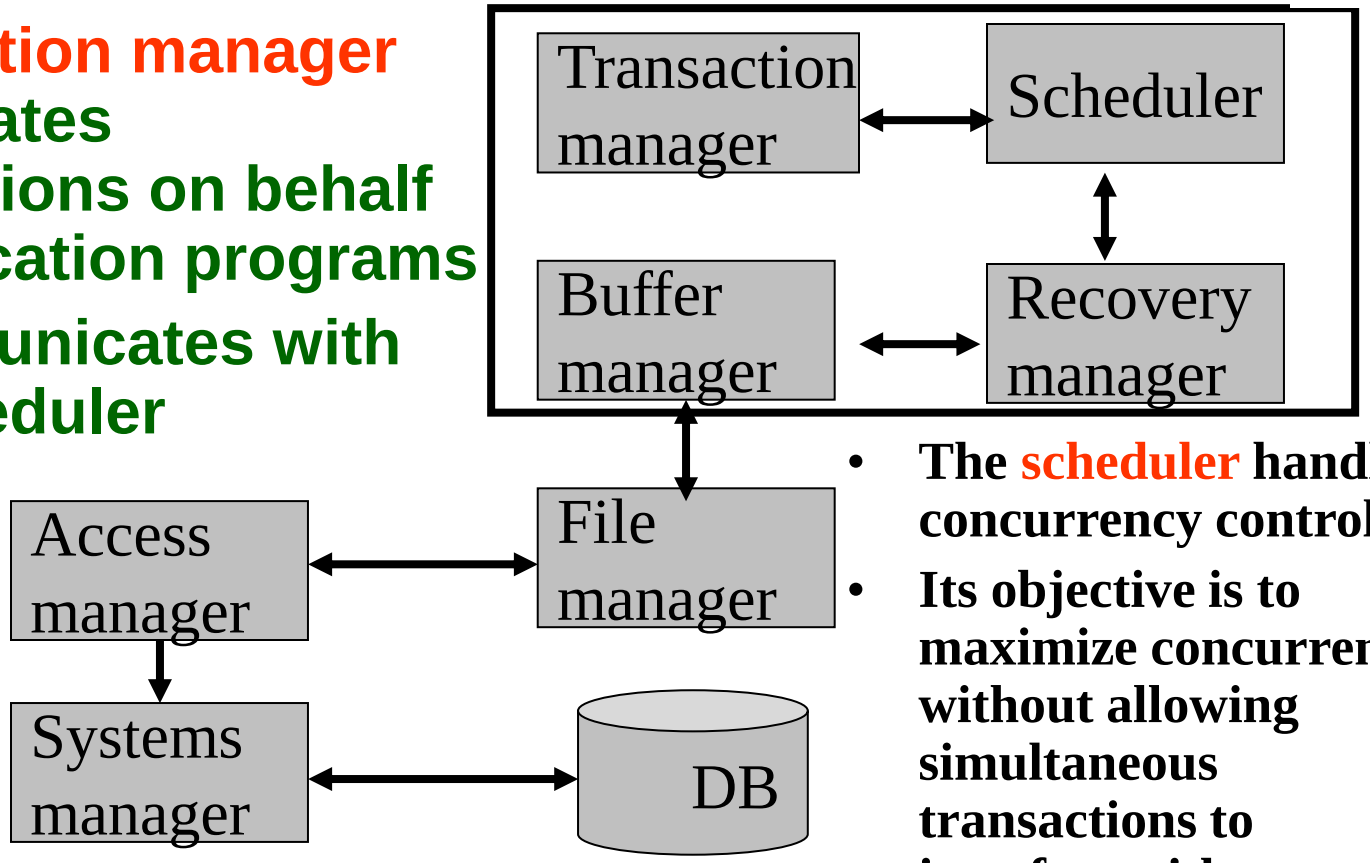
- **Each local DBMS might export:**
  - **stored procedures or**
  - **an SQL interface.**
- **Operations at each site are grouped together as a subtransaction and the site is referred to as a cohort of the distributed transaction**
  - **Each subtransaction is treated as a transaction at its site**
- **Coordinator module (part of TP monitor) supports ACID properties of distributed transaction**
  - **Transaction manager acts as coordinator**

Transaction Processing Monitor
or
TP Monitor



DBMS Transaction Subsystem

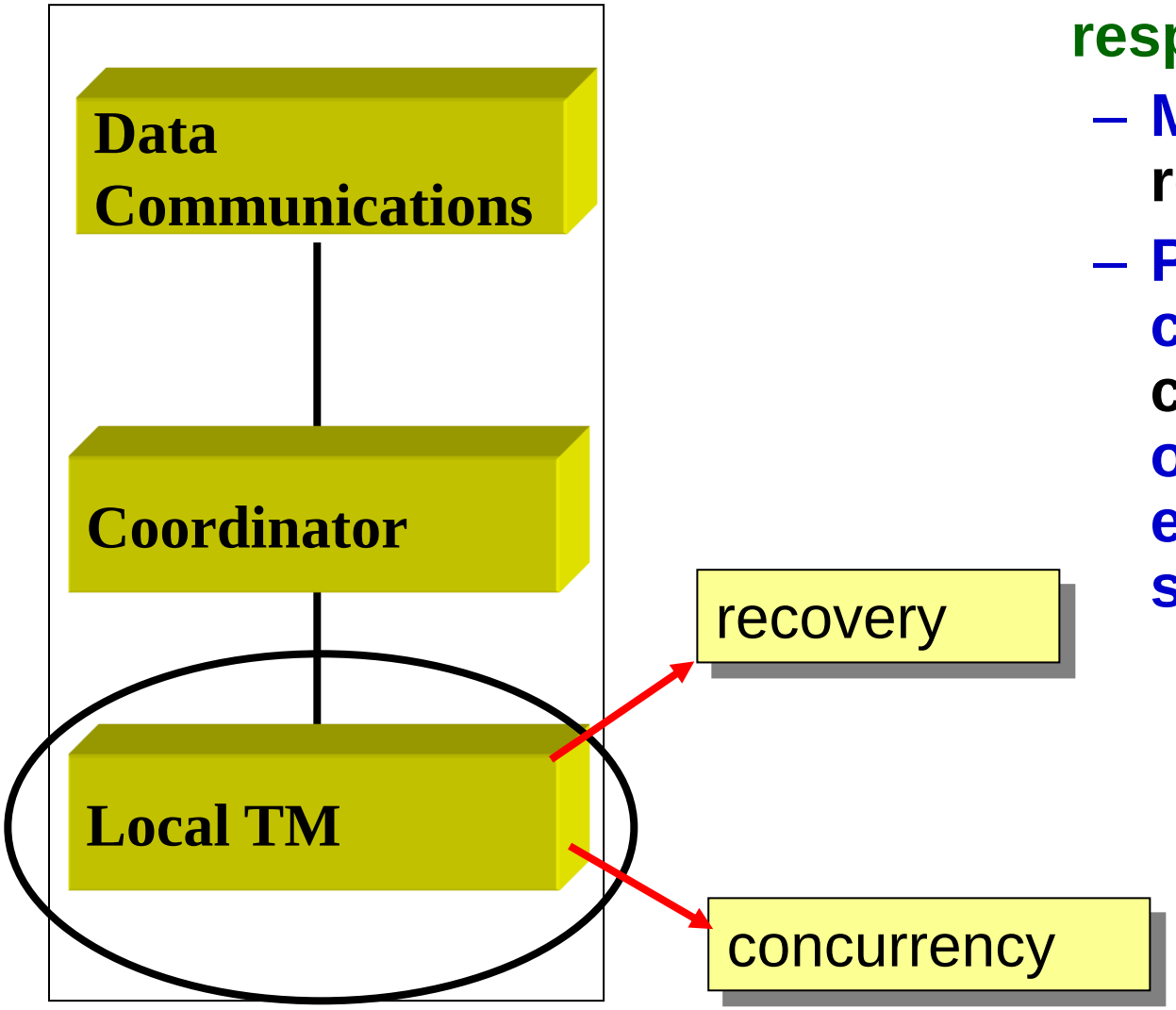
- **Transaction manager coordinates transactions on behalf of application programs**
- **It communicates with the scheduler**



- **The scheduler handles concurrency control.**
- **Its objective is to maximize concurrency without allowing simultaneous transactions to interfere with one another**

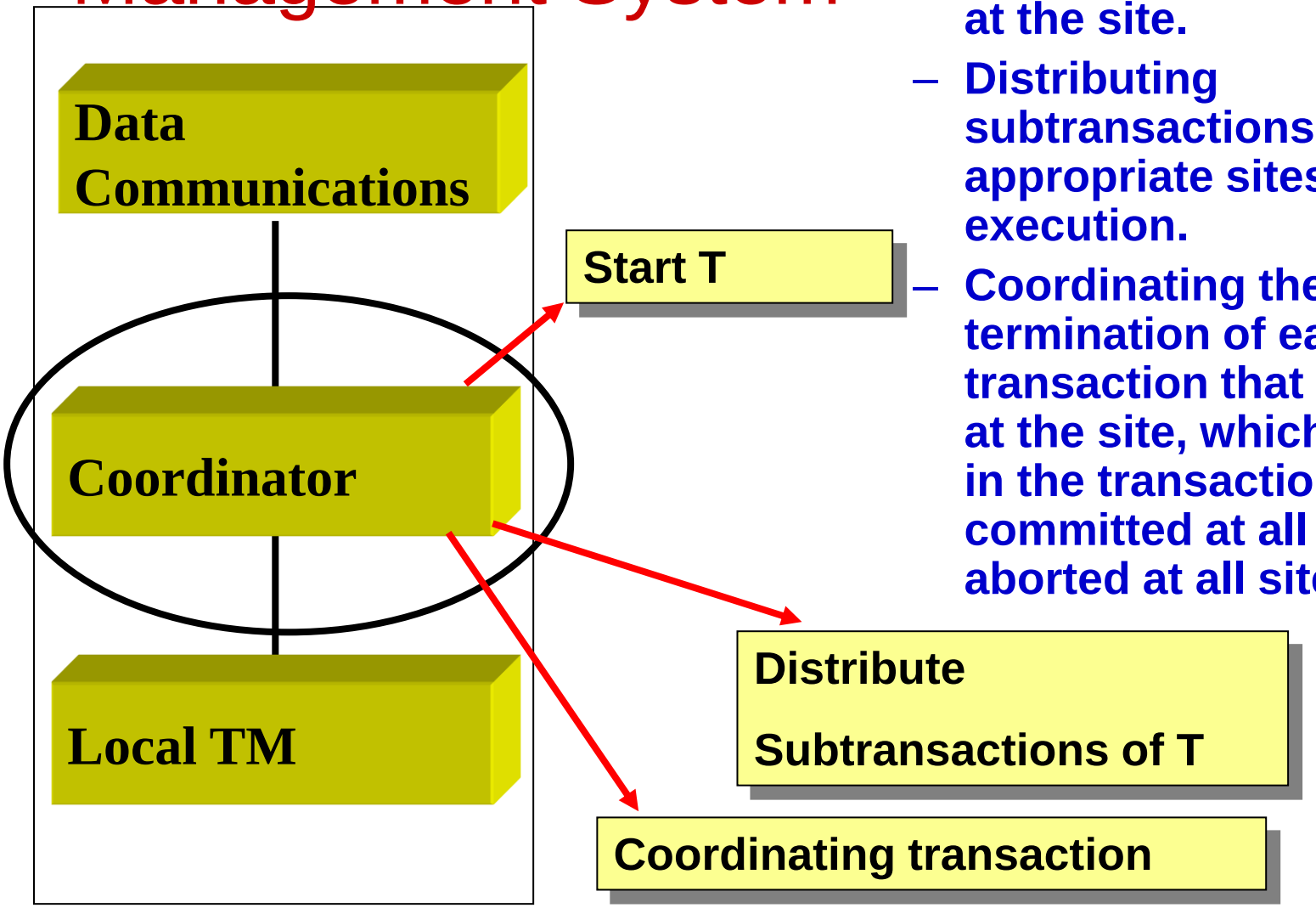
- The **recovery manager** ensures that the database is restored to the right state before a failure occurred.
- The **buffer manager** is responsible for the transfer of data between disk storage and main memory.

Distributed Transactions Management System



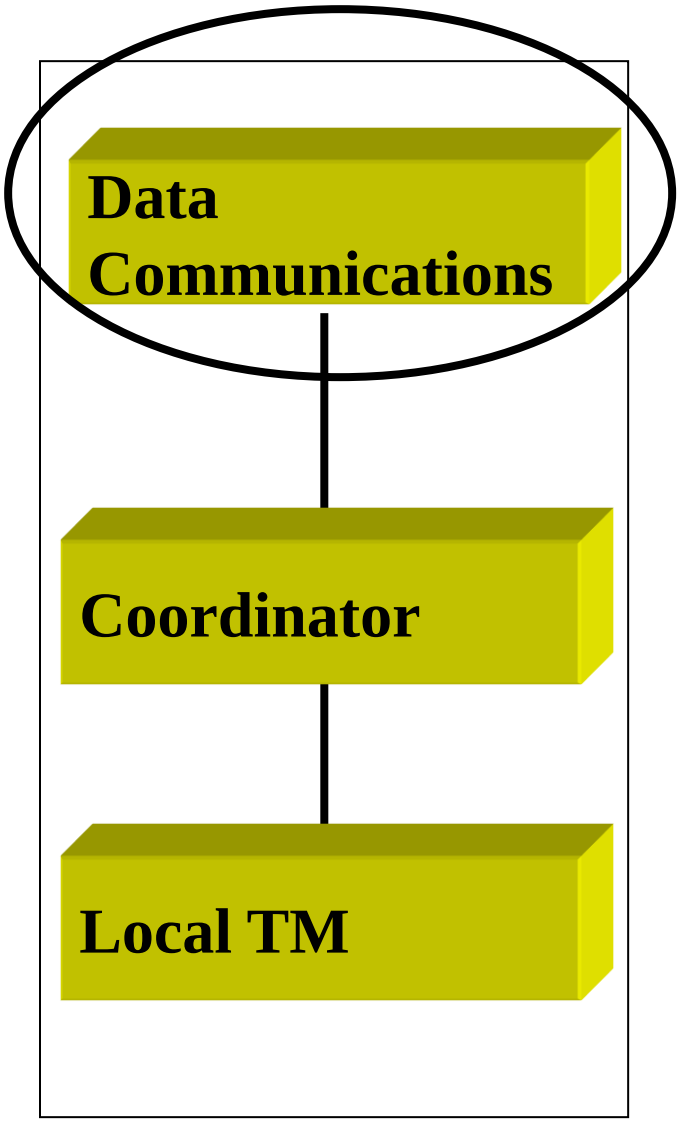
- **Each site has a local transaction manager responsible for:**
  - **Maintaining a log for recovery purposes**
  - **Participating in coordinating the concurrent execution of the transactions executing at that site.**

Distributed Transactions Management System



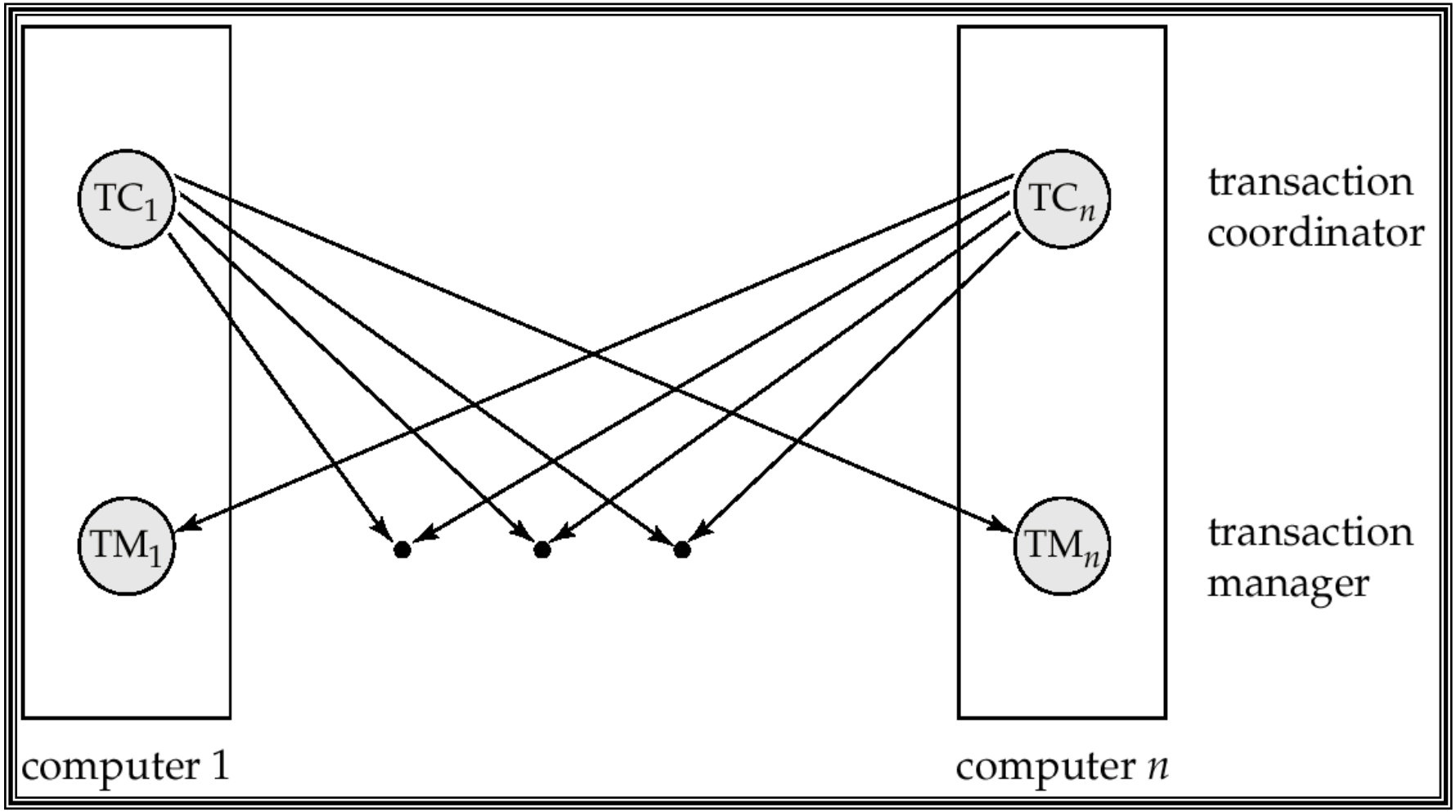
- Each site has a transaction coordinator, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

Distributed Transactions Management System

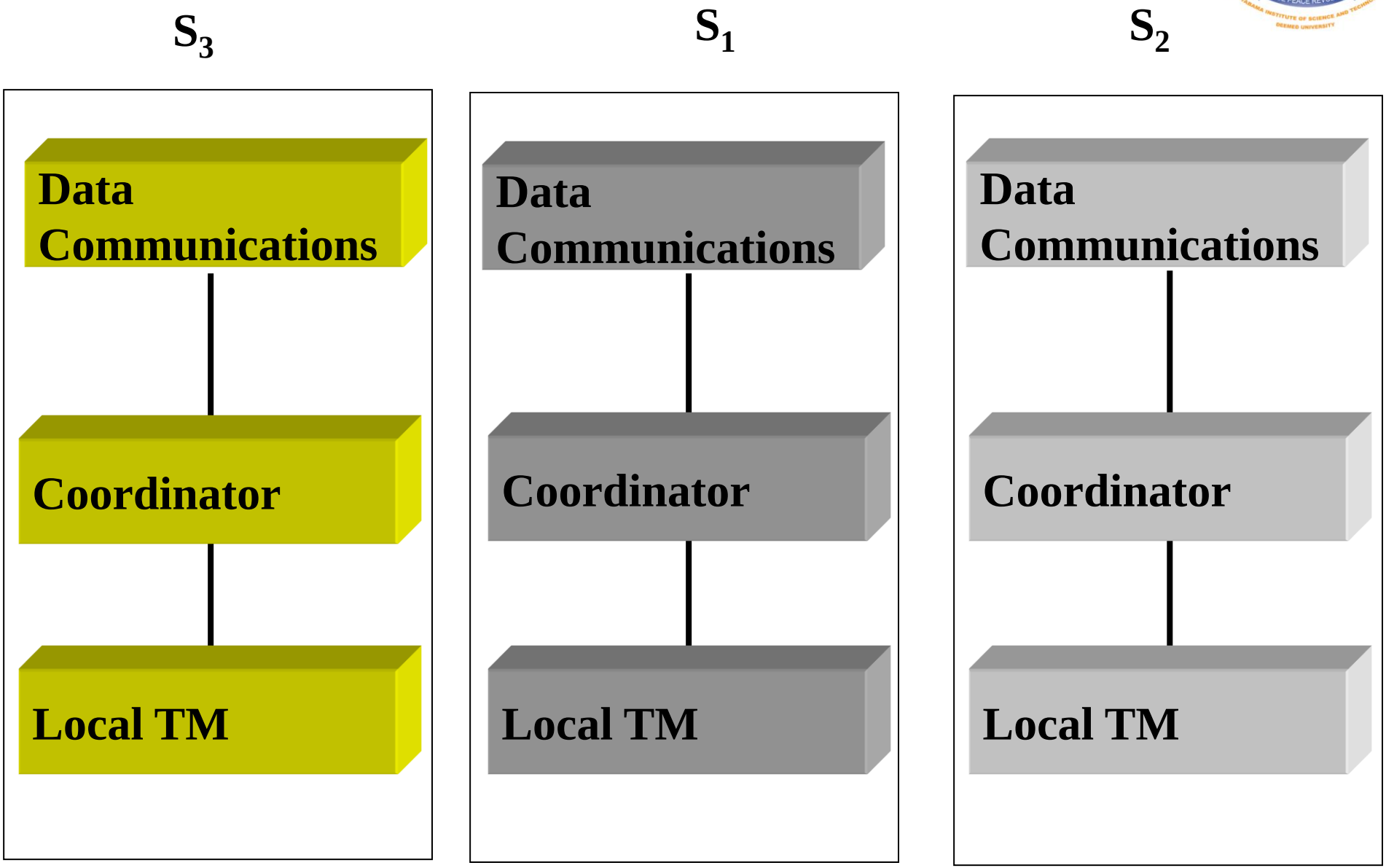


- **Each site has a data communications module that handle inter-site communications**
- **TMs at each site do not communicate with each other directly**

Transaction System Architecture



Coordination of Distributed Transaction



Coordination of Distributed Transaction

S_3 S_1 S_2

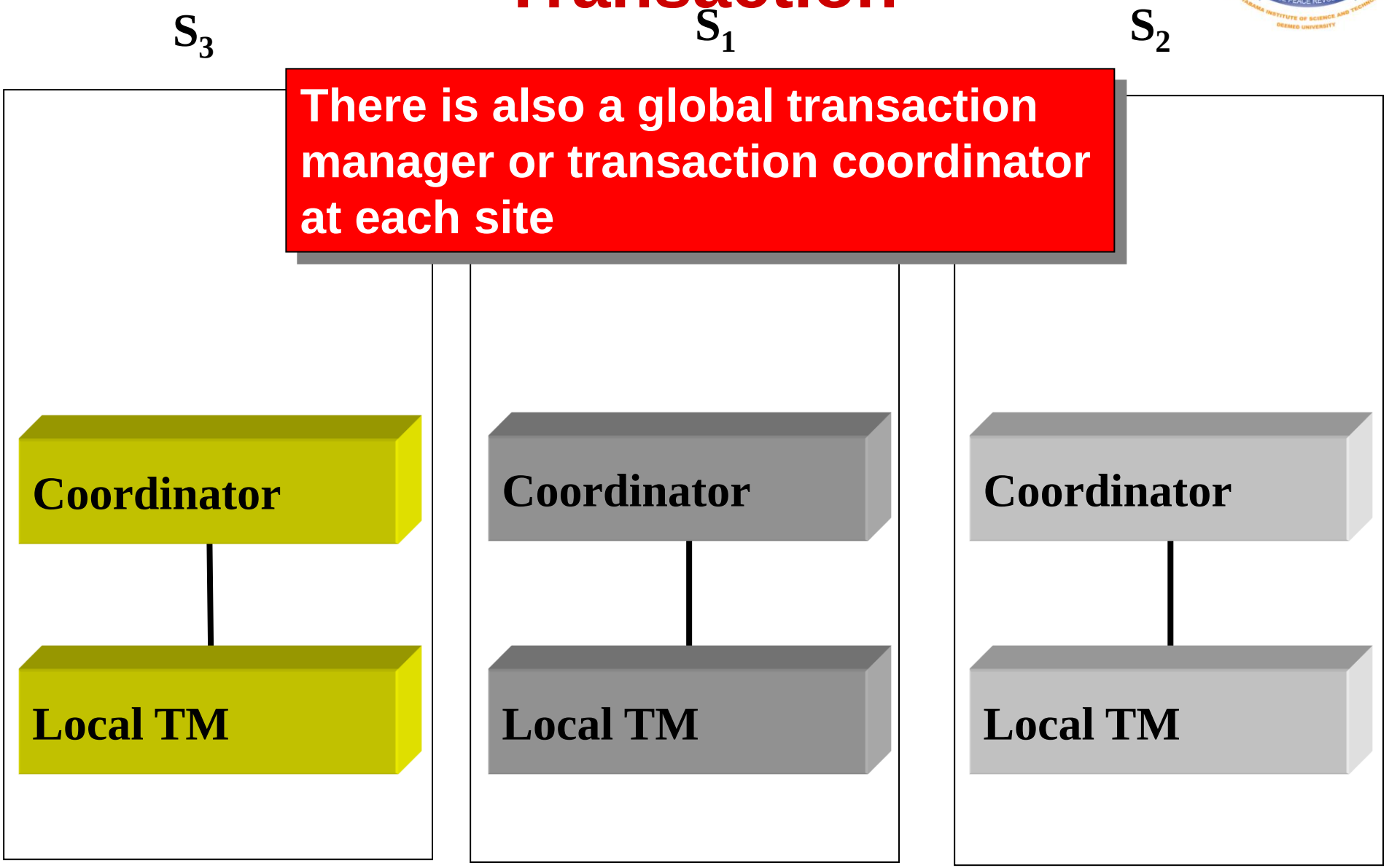
In a DDBMS, the local transaction Manager (TM) still exists in each local DBMS

Local TM

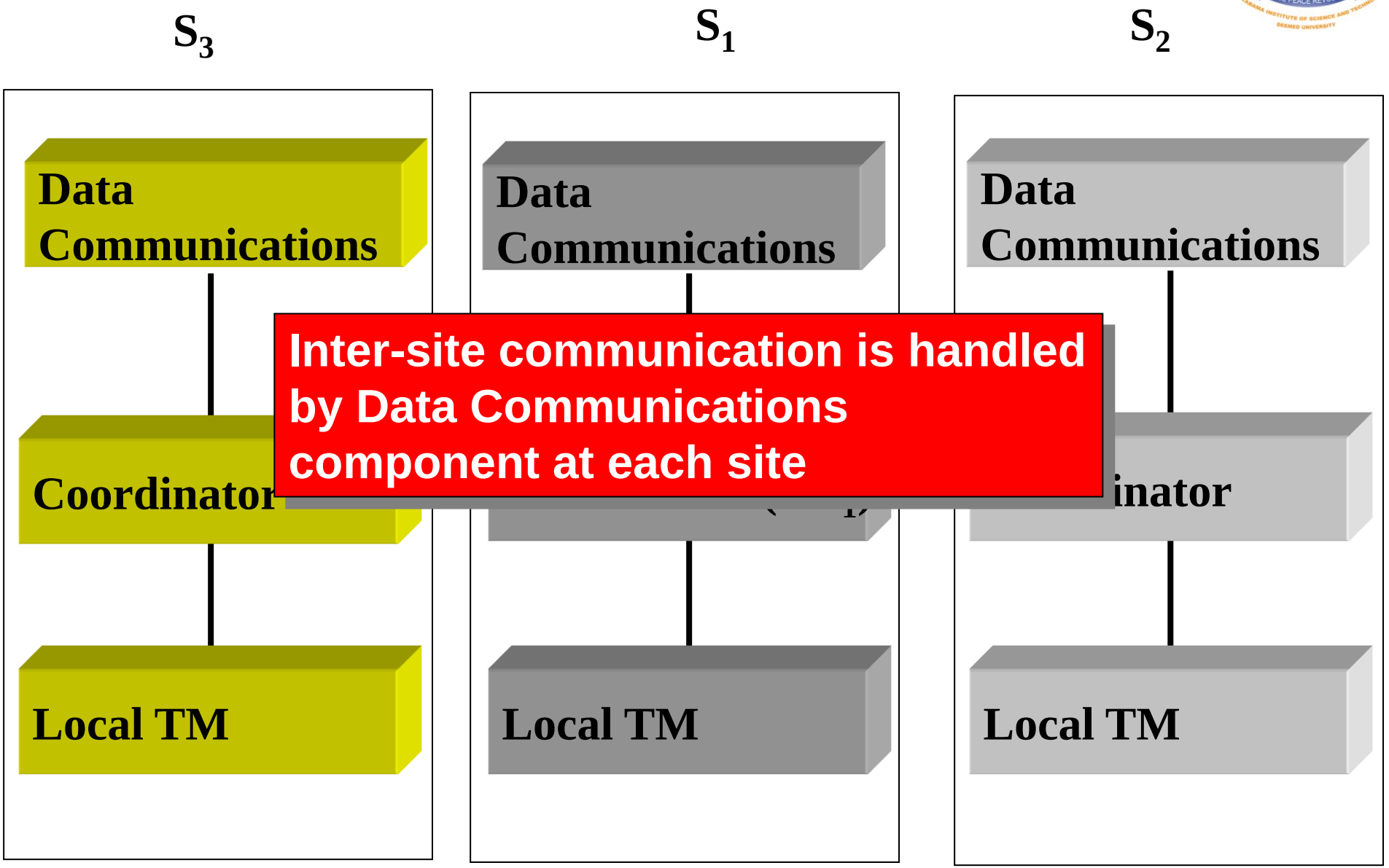
Local TM

Local TM

Coordination of Distributed Transaction



Coordination of Distributed Transaction



Procedure: Coordination of Distributed Transaction

- **Fragmented schema: S_1, S_2, S_{21}, S_{22} , S_{23} etc.,**
- **A transaction T that prints out the names of all staff;**
- **Subtransactions:**
  - **T_{s3} : at site 3**
  - **T_{s5} : at site 5**
  - **T_{s7} : at site 7**

S_1: $\prod_{\text{Sno, position,sex, dob, salary,nin}}$ (Staff) site5

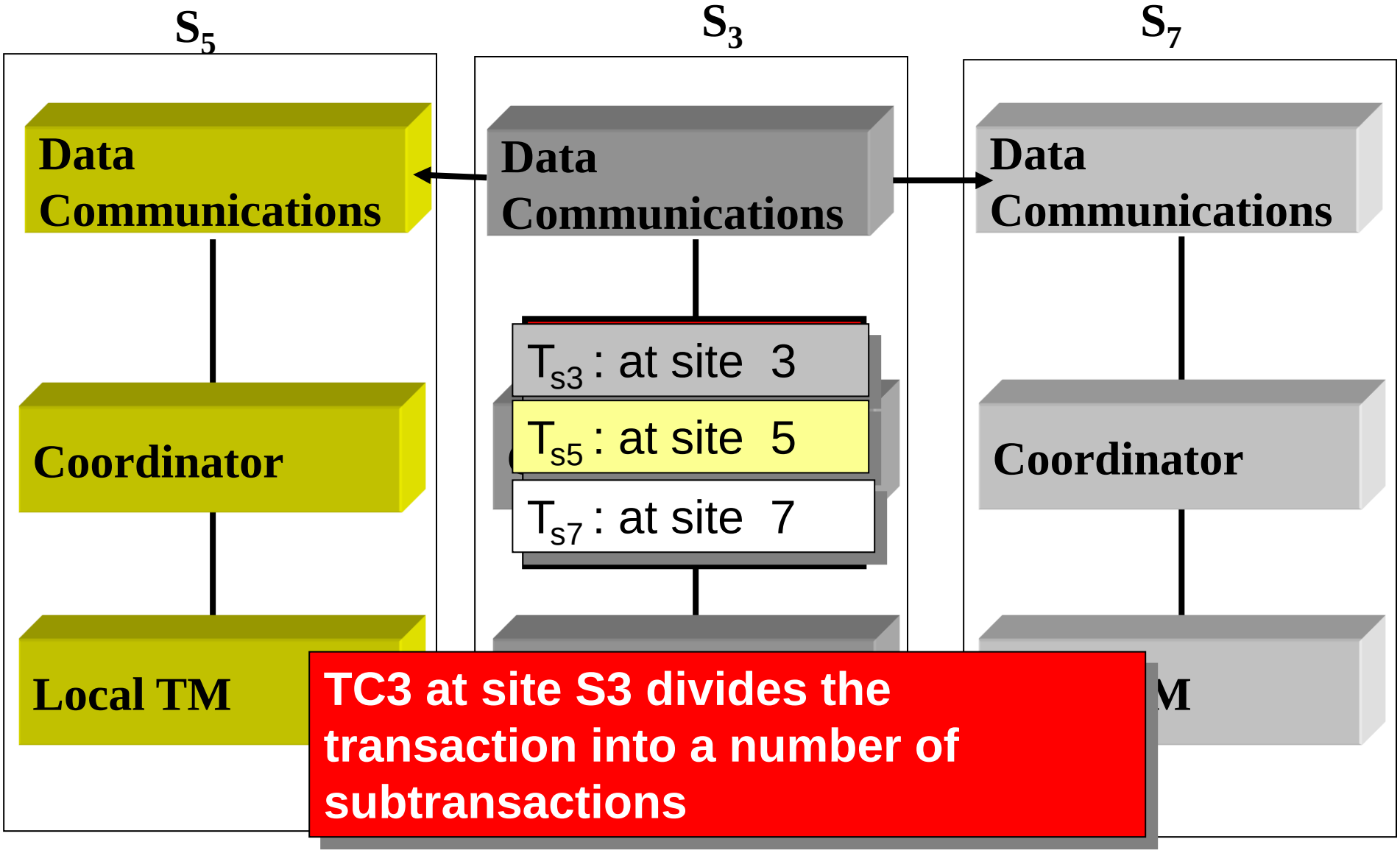
S_2: $\prod_{\text{Sno, fname, lname,address,telno,bno}}$ (Staff)

S_{21}: $\delta_{\text{bno='B3'}}(S_2)$ site 3

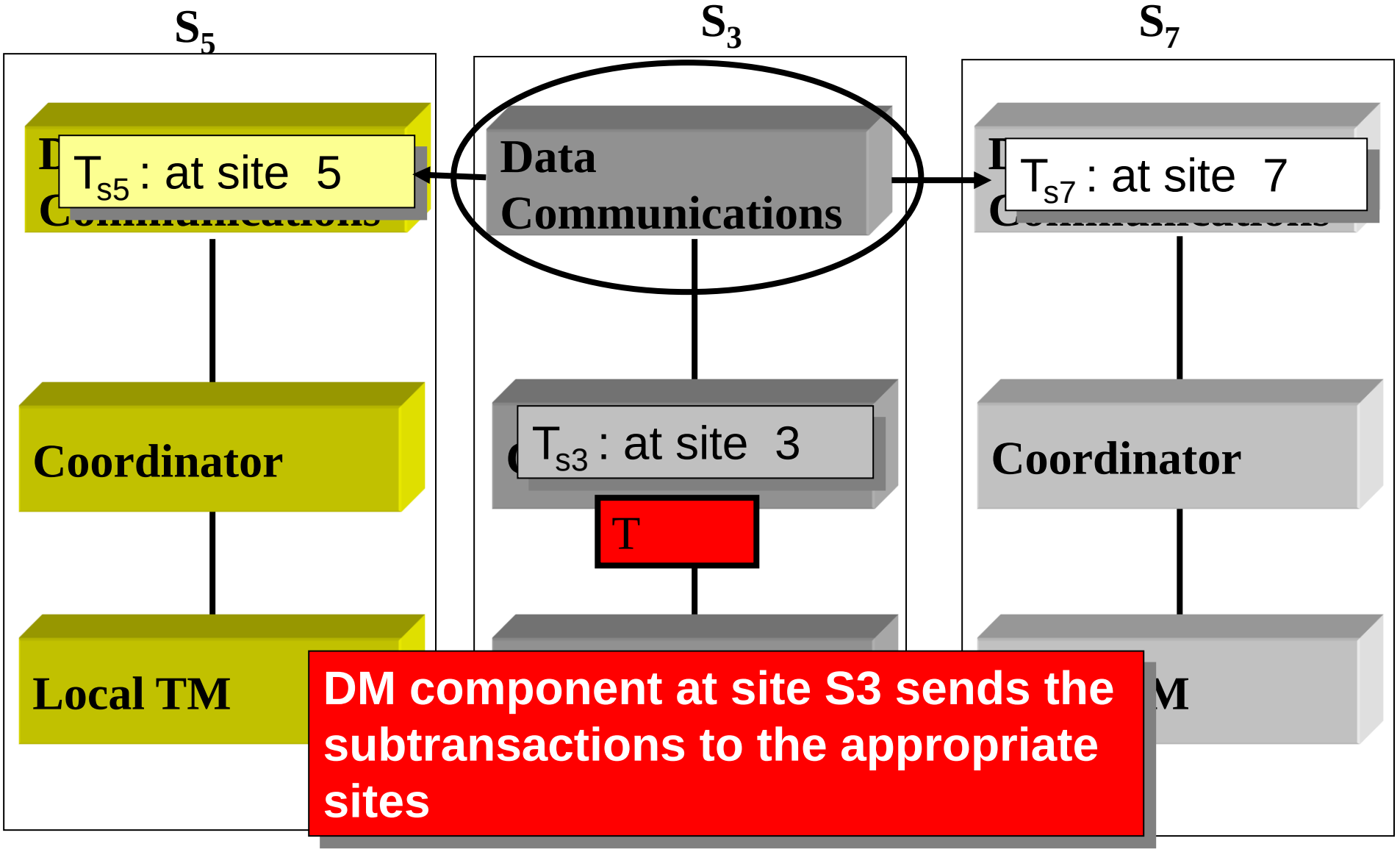
S_{22}: $\delta_{\text{bno='B5'}}(S_2)$ site 5

S_{22}: $\delta_{\text{bno='B7'}}(S_2)$ site 7

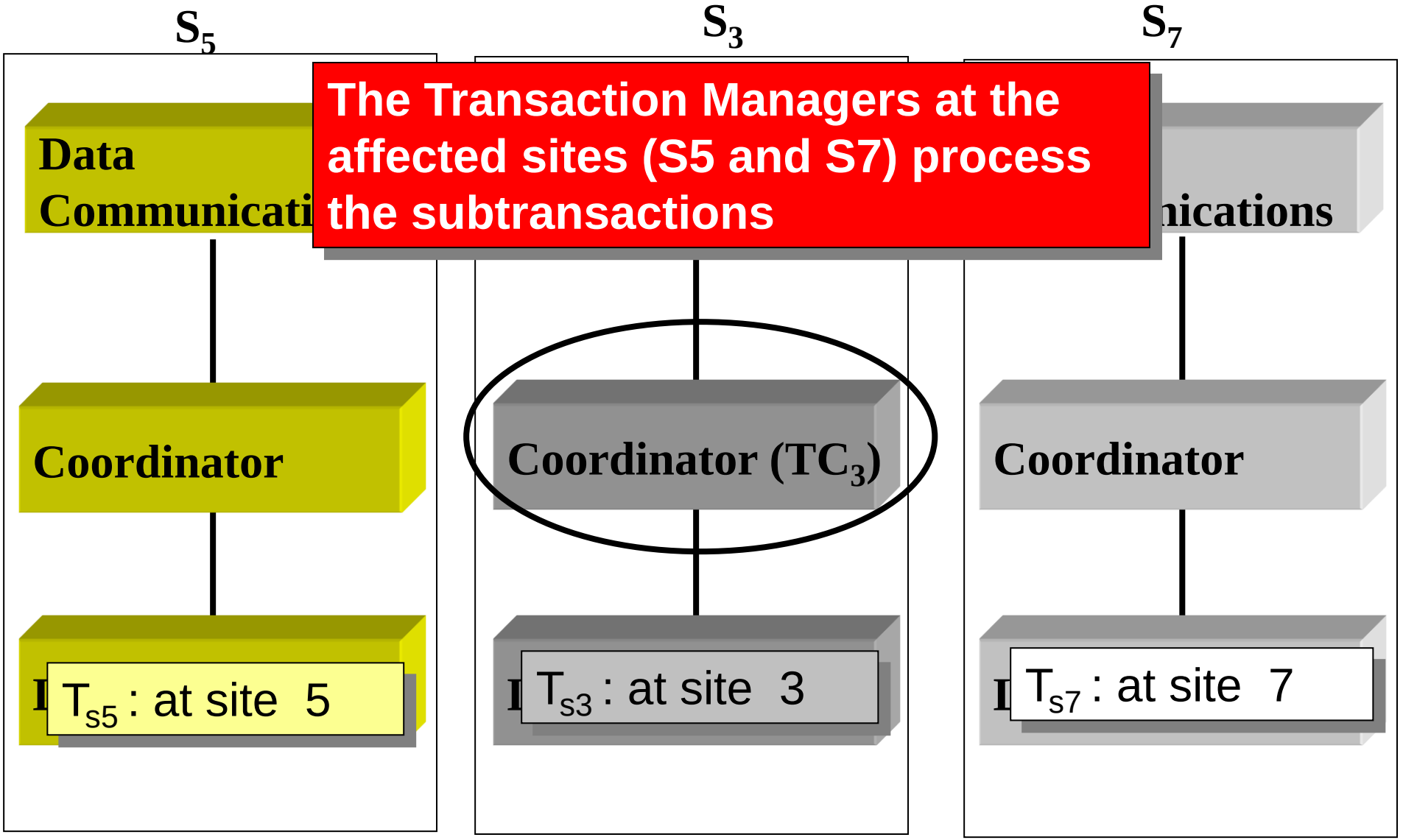
Procedure: Coordination of Distributed Transaction



Procedure: Coordination of Distributed Transaction



Procedure: Coordination of Distributed Transaction

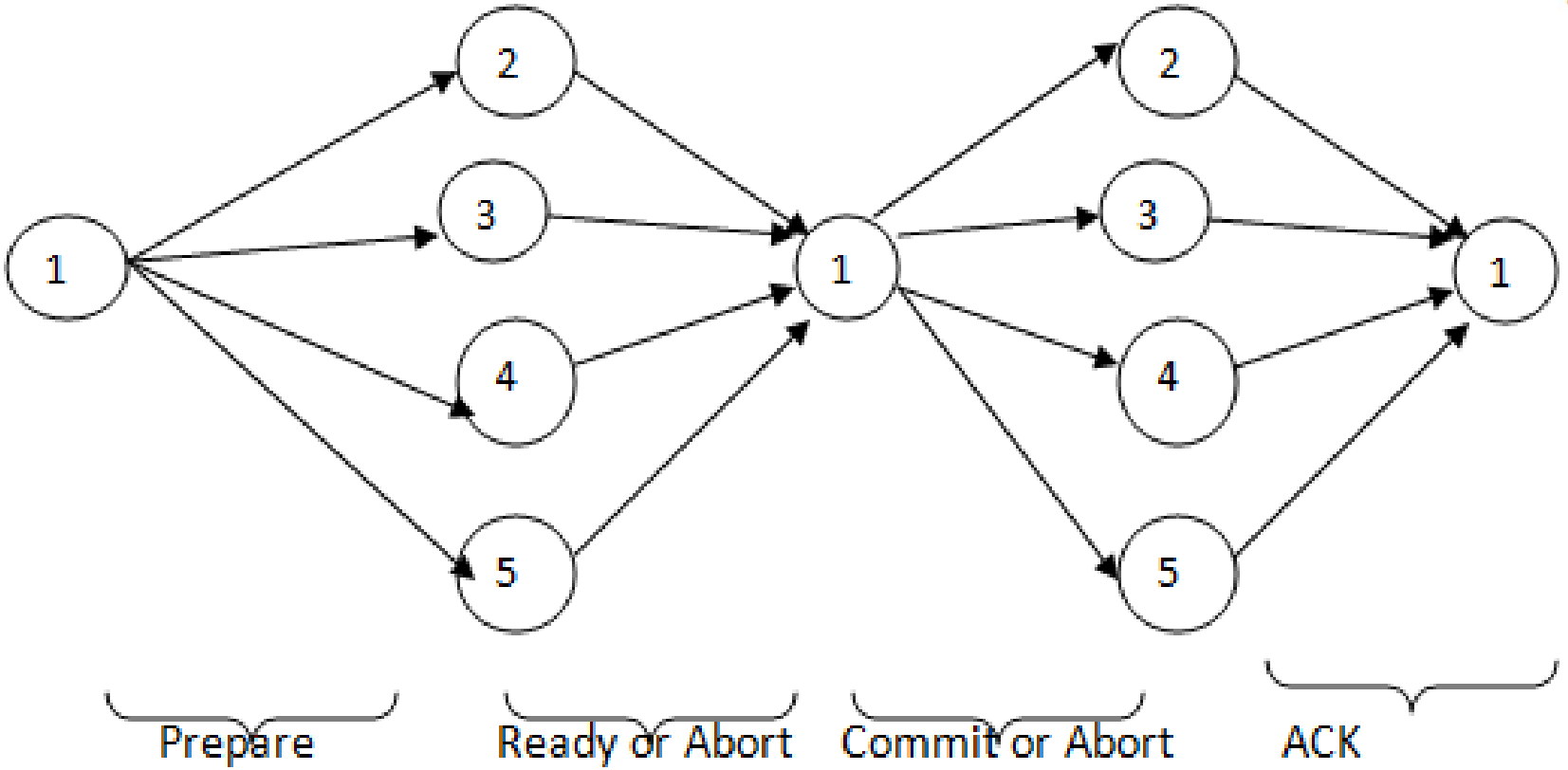


Communication structure for commit protocols

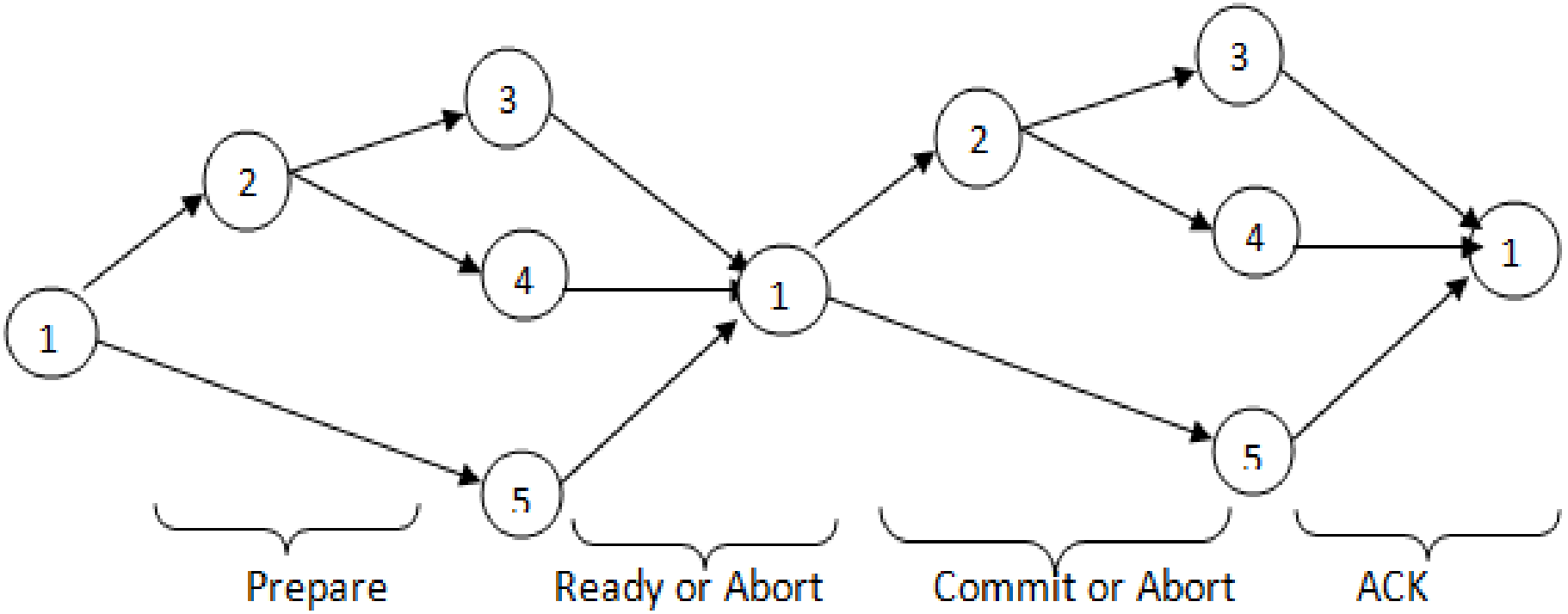
The various structure in which it is implemented are

1. **Centralized**
2. **Hierarchical**
3. **Linear**
4. **Distributed**

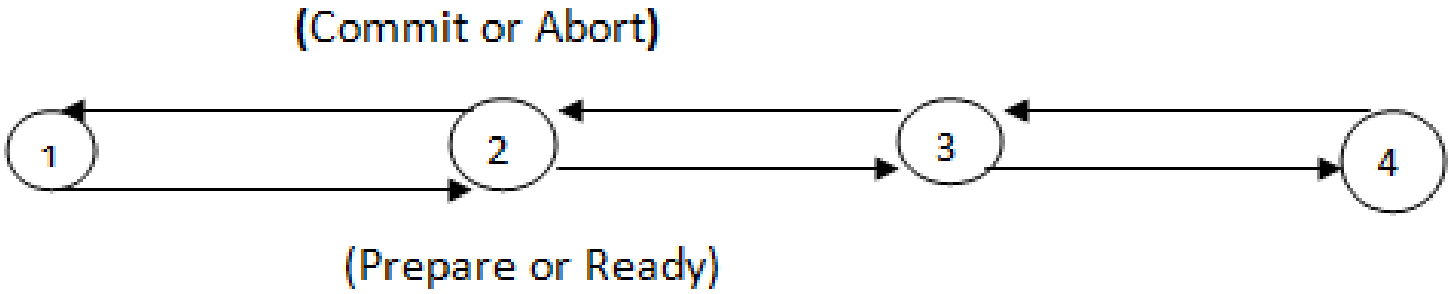
Centralized



Hierarchical

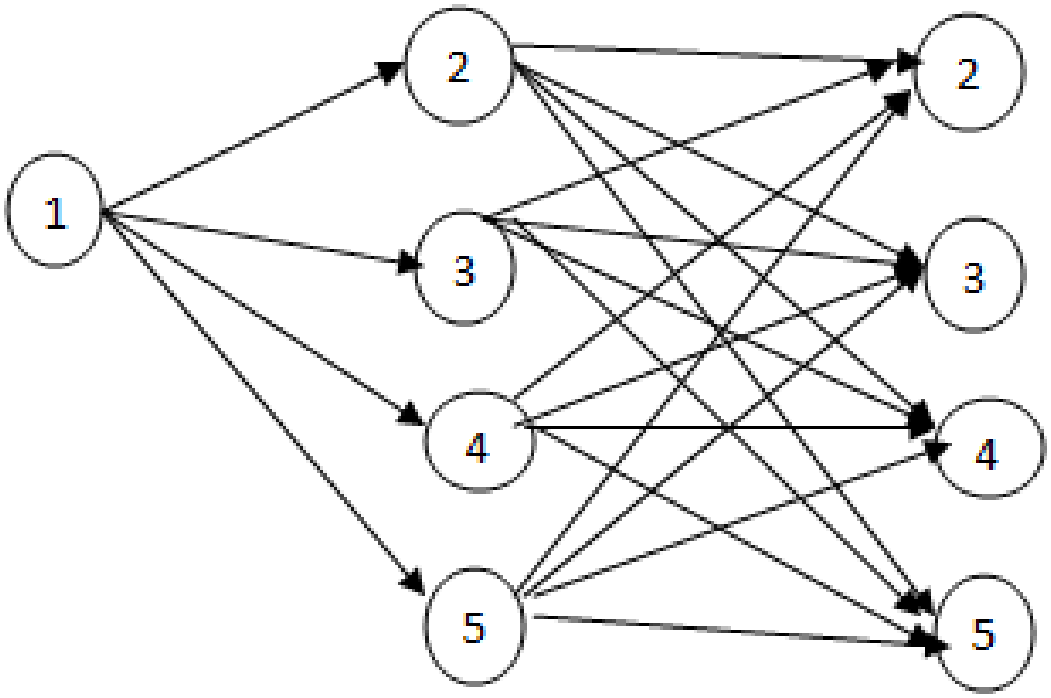


Linear



Ordering is defined

Distributed



Prepare

Ready or Abort

(No messages are required for the decision)

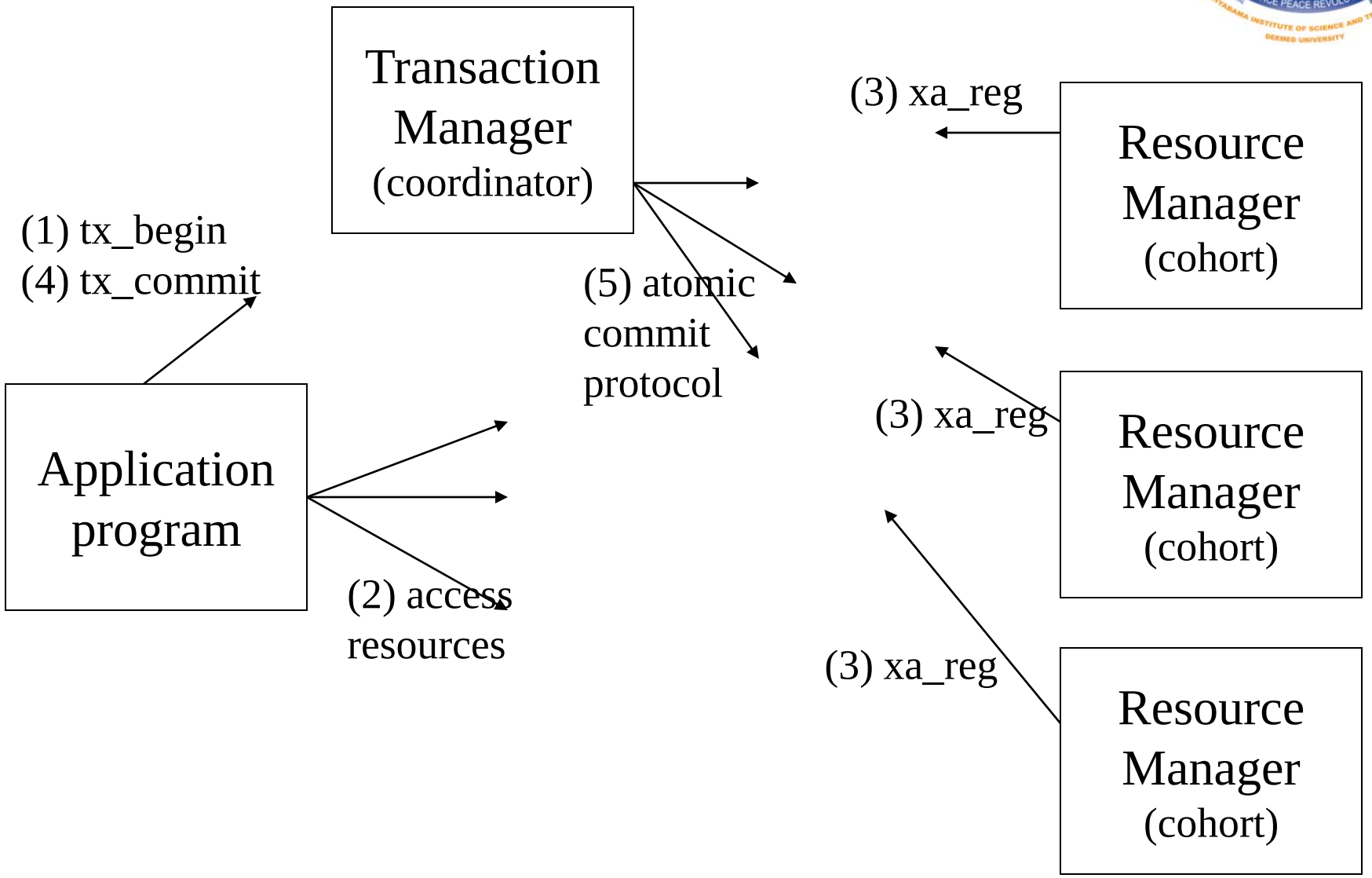
ACID Properties

- **Each local DBMS:**
  - **Supports ACID locally for each subtransaction**
    - **Just like any other transaction that executes there**
  - **Eliminates local deadlocks.**
- **The additional issues are:**
  - **Global atomicity: all cohorts must abort or all commit**
  - **Global deadlocks: there must be no deadlocks involving multiple sites**
  - **Global serialization: distributed transaction must be globally serializable**

Global Atomicity

- All subtransactions of a distributed transaction must commit or all must abort
- An atomic commit protocol, initiated by a coordinator (*e.g.*, the transaction manager), ensures this.
  - Coordinator polls cohorts to determine if they are all willing to commit
- Protocol is supported in the xa interface between a transaction manager and a resource manager

Atomic Commit Protocol



Cohort Abort

- **Why might a cohort abort?**
  - **Deferred evaluation** of integrity constraints
  - **Validation failure** (optimistic control)
  - **Deadlock**
  - **Crash** of cohort site
  - **Failure** prevents communication with cohort site

Atomic Commit Protocol

- **Two-phase commit protocol:** most commonly used atomic commit protocol.
- **Implemented as:** an exchange of messages between the coordinator and the cohorts.
- **Guarantees global atomicity:** of the transaction even if failures should occur while the protocol is executing.

Two-Phase Commit (The Transaction Record)

- **During the execution of the transaction, before the two-phase commit protocol begins:**
  - **When the application calls tx_begin to start the transaction, the coordinator creates a transaction record for the transaction in volatile memory**
  - **Each time a resource manager calls xa_reg to join the transaction as a cohort, the coordinator appends the cohort's identity to the transaction record**

Two-Phase Commit -- Phase 1

- When application invokes tx_commit, coordinator
- Sends prepare message (coordin. to all cohorts) :
  - If cohort wants to abort at any time prior to or on receipt of the message, it aborts and releases locks
  - If cohort wants to commit, it moves all update records to mass store by forcing a prepare record to its log
    - Guarantees that cohort will be able to commit (despite crashes) if coordinator decides commit (since update records are durable)
    - Cohort enters prepared state
  - Cohort sends a vote message ("ready" or "aborting"). It
    - cannot change its mind
    - retains all locks if vote is "ready"
    - enters uncertain period (it cannot foretell final outcome)

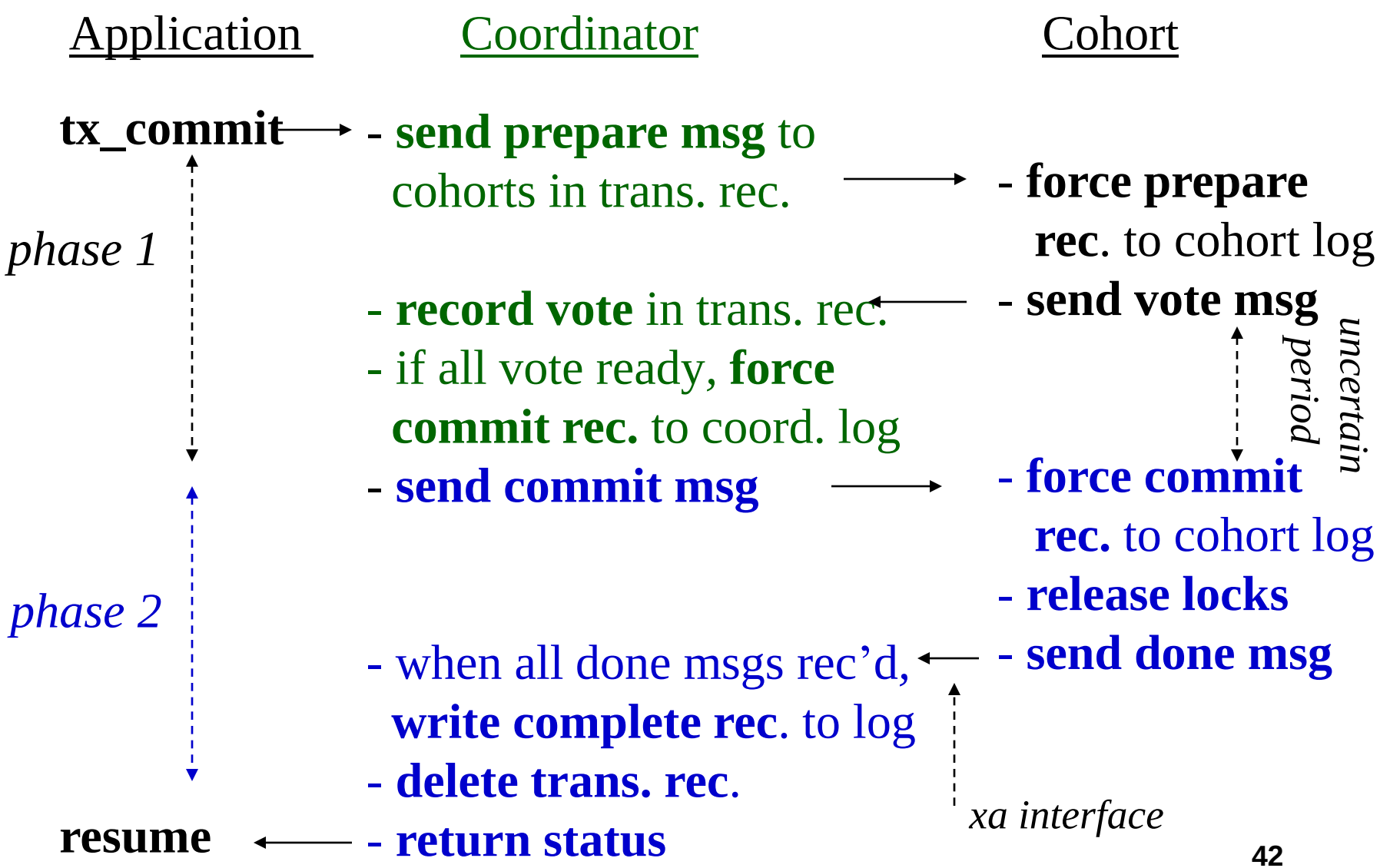
Two-Phase Commit -- Phase 1

- **Vote message** (cohort to coordinator): **Cohort** indicates it is **"ready"** to commit or is **"aborting"**
  - **Coordinator records vote** in transaction record
  - **If any votes are "aborting",** coordinator decides abort **and** deletes transaction record
  - **If all are "ready",** coordinator decides **commit,** forces **commit record** (containing transaction record) to its log **(end of phase 1)**
    - **Transaction committed when** commit record is durable
    - **Since all cohorts are in prepared state,** transaction can be committed despite any failures
  - **Coordinator sends commit** or **abort message** to all cohorts

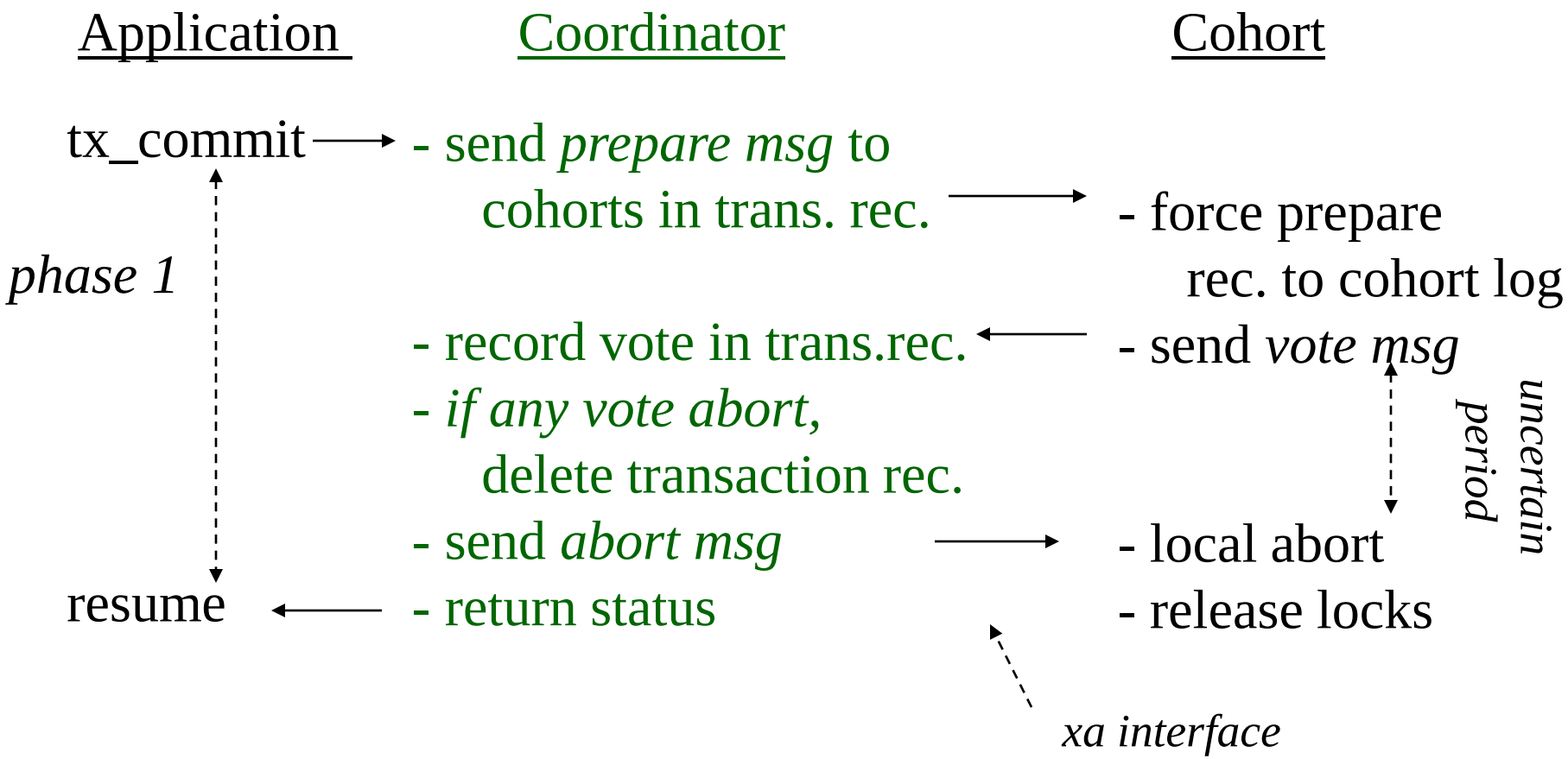
Two-Phase Commit -- Phase 2

- **Commit or abort message (coordinator to cohort):**
  - **If commit message**
    - **cohort commits locally by forcing a commit record to its log**
    - **cohort sends done message to coordinator**
  - **If abort message, it aborts**
  - **In either case, locks are released and uncertain period ends**
- **Done message (cohort to coordinator):**
  - **When coordinator receives a done message from each cohort,**
    - **it writes a complete record to its log and**
    - **deletes transaction record from volatile store**

Two-Phase Commit (commit case)



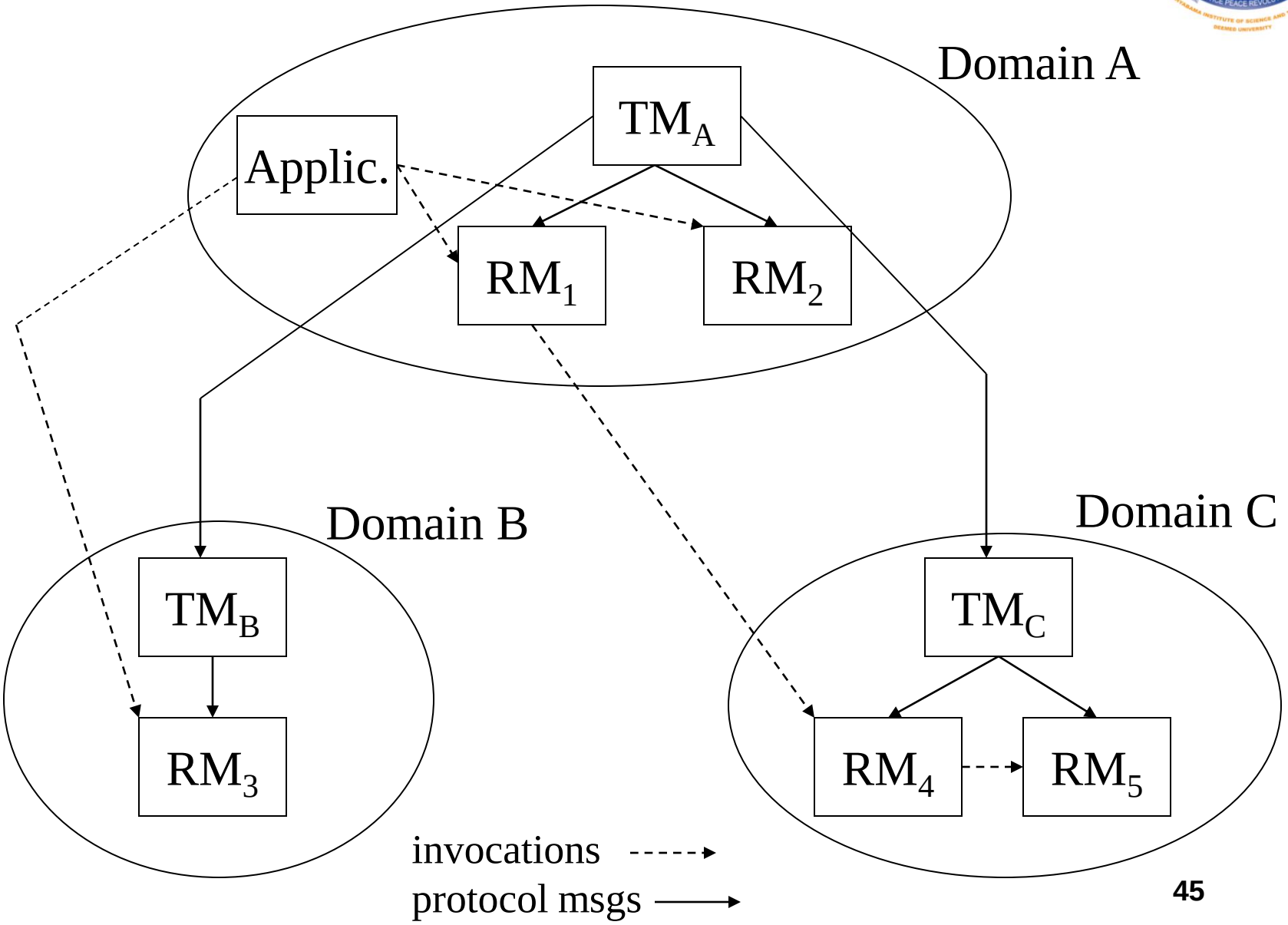
Two-Phase Commit (abort case)



Distributing the Coordinator

- A transaction manager controls resource managers in its domain
- When a cohort in domain A invokes a resource manager RM_B in domain B:
  - The local transaction manager TM_A and remote transaction manager TM_B are notified
  - TM_B is a cohort of TM_A and a coordinator of RM_B
- A coordinator/cohort tree results

Coordinator/Cohort Tree



Distributing the Coordinator

- **The two-phase commit protocol progresses down and up the tree in each phase**
  - **When TM_B gets a *prepare msg* from TM_A it sends a *prepare msg* to each child and waits**
  - **If each child votes ready, TM_B sends a *ready msg* to TM_A**
    - **if not it sends an *abort msg***

Failures and Two-Phase Commit

- **A participant recognizes two failure situations.**
  - **Timeout :** No response to a message. Execute a timeout protocol
  - **Crash :** On recovery, execute a restart protocol
- **If a cohort cannot complete the protocol until some failure is repaired, it is said to be blocked**
  - **Blocking** can impact performance at the cohort site since locks cannot be released

Timeout Protocol

- **Cohort times out waiting for *prepare message***
  - **Abort the subtransaction**
    - **Since the (distributed) transaction cannot commit unless cohort votes to commit, atomicity is preserved**
- **Coordinator times out waiting for *vote message***
  - **Abort the transaction**
    - **Since coordinator controls decision, it can force all cohorts to abort, preserving atomicity**

Timeout Protocol

- **Cohort (in prepared state) times out waiting for commit/abort message**
  - **Cohort is blocked since it does not know coordinator's decision**
    - **Coordinator might have decided commit or abort**
    - **Cohort cannot unilaterally decide since its decision might be contrary to coordinator's decision, violating atomicity**
    - **Locks cannot be released**
  - **Cohort requests status from coordinator; remains blocked**
- **Coordinator times out waiting for done message**
  - **Requests done message from delinquent cohort**

Restart Protocol - Cohort

- **On restart cohort finds in its log:**
  - **begin_transaction record, but no prepare record:**
    - **Abort (transaction cannot have committed because cohort has not voted)**
  - **prepare record, but no commit record (cohort crashed in its uncertain period)**
    - **Does not know if transaction committed or aborted**
    - **Locks items mentioned in update records before restarting system**
    - **Requests status from coordinator and blocks until it receives an answer**
  - **commit record**
    - **Recover transaction to committed state using log**

Restart Protocol - Coordinator

- **On restart:**
  - **Search log and restore to volatile memory the transaction record of each transaction for which there is a commit record, but no complete record**
    - **Commit record contains transaction record**
- **On receiving a request from a cohort for transaction status:**
  - **If transaction record exists in volatile memory, reply based on information in transaction record**
  - **If no transaction record exists in volatile memory, reply abort**
    - **Referred to as presumed abort property**

Presumed Abort Property

- If when a cohort asks for the status of a transaction there is no transaction record in coordinator's volatile storage, either
  - The coordinator had aborted the transaction and deleted the transaction record
  - The coordinator had crashed and restarted and did not find the commit record in its log because
    - It was in Phase 1 of the protocol and had not yet made a decision, or
    - It had previously aborted the transaction

Presumed Abort Property

- or
  - The coordinator had crashed and restarted and found a complete record for the transaction in its log
  - The coordinator had committed the transaction, received done messages from all cohorts and hence deleted the transaction record from volatile memory
- The last two possibilities cannot occur
  - In both cases, the cohort has sent a done message and hence would not request status
- Therefore, coordinator can respond abort

Heuristic Commit

- **What does a cohort do when in the blocked state and the coordinator does not respond to a request for status?**
  - **Wait until the coordinator is restarted**
  - **Give up, make a unilateral decision, and attach a fancy name to the situation.**
    - **Always abort**
    - **Always commit**
    - **Always commit certain types of transactions and always abort others**
  - **Resolve the potential loss of atomicity outside the system**
    - **Call on the phone or send email**

Variants/Optimizations

- **Read-only subtransactions need not participate in the protocol as cohorts**
  - **As soon as such a transaction receives the prepare message, it can give up its locks and exit the protocol.**
- **Transfer of coordination**

Transfer of Coordination

- Sometimes it is not appropriate for the coordinator (in the initiator's domain) to coordinate the commit
  - Perhaps the initiator's domain is a convenience store and the bank does not trust it to perform the commit
- Ability to coordinate the commit can be transferred to another domain
  - Linear commit
  - Two-phase commit without a prepared state

Linear Commit

- **Variation of two-phase commit that involves transfer of coordination**
- **Used in a number of Internet commerce protocols**
- **Cohorts are assumed to be connected in a linear chain**

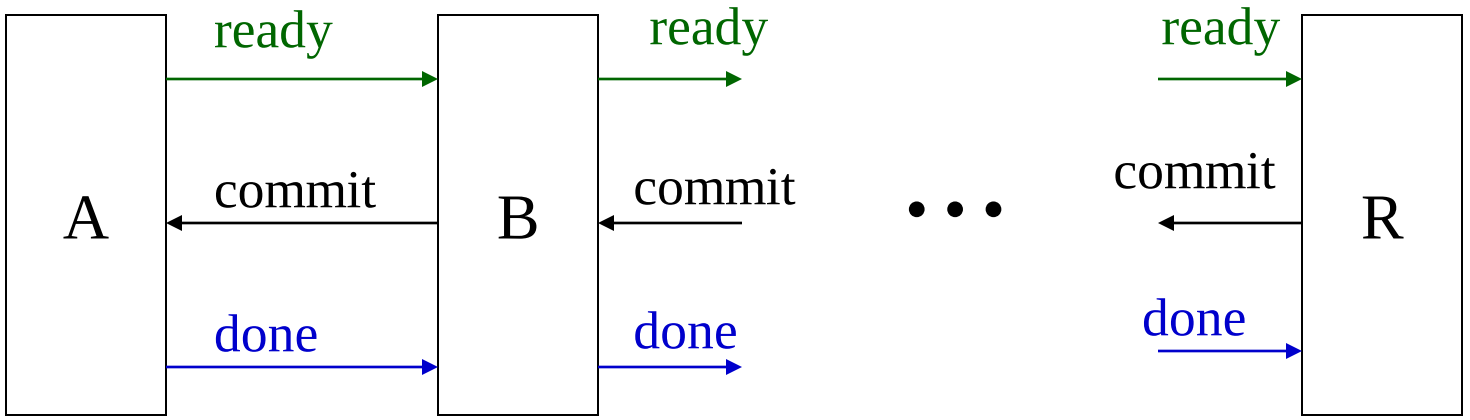
Linear Commit Protocol

- When leftmost cohort A is ready to commit it goes into a prepared state and sends a vote message ("ready") to the cohort to its right B (requesting B to act as coordinator).
- After receiving the vote message, if B is ready to commit, it also goes into a prepared state and sends a vote message ("ready") to the cohort to its right R (requesting R to act as coordinator)
- And so on ...

Linear Commit Protocol

- When vote message reaches the rightmost cohort R
  - If R is ready to commit, it commits the entire transaction (acting as coordinator) and sends a commit message to the cohort on its left
- The commit message propagates down the chain until it reaches A
- When A receives the commit message it sends a done message to B that also propagates

Linear Commit



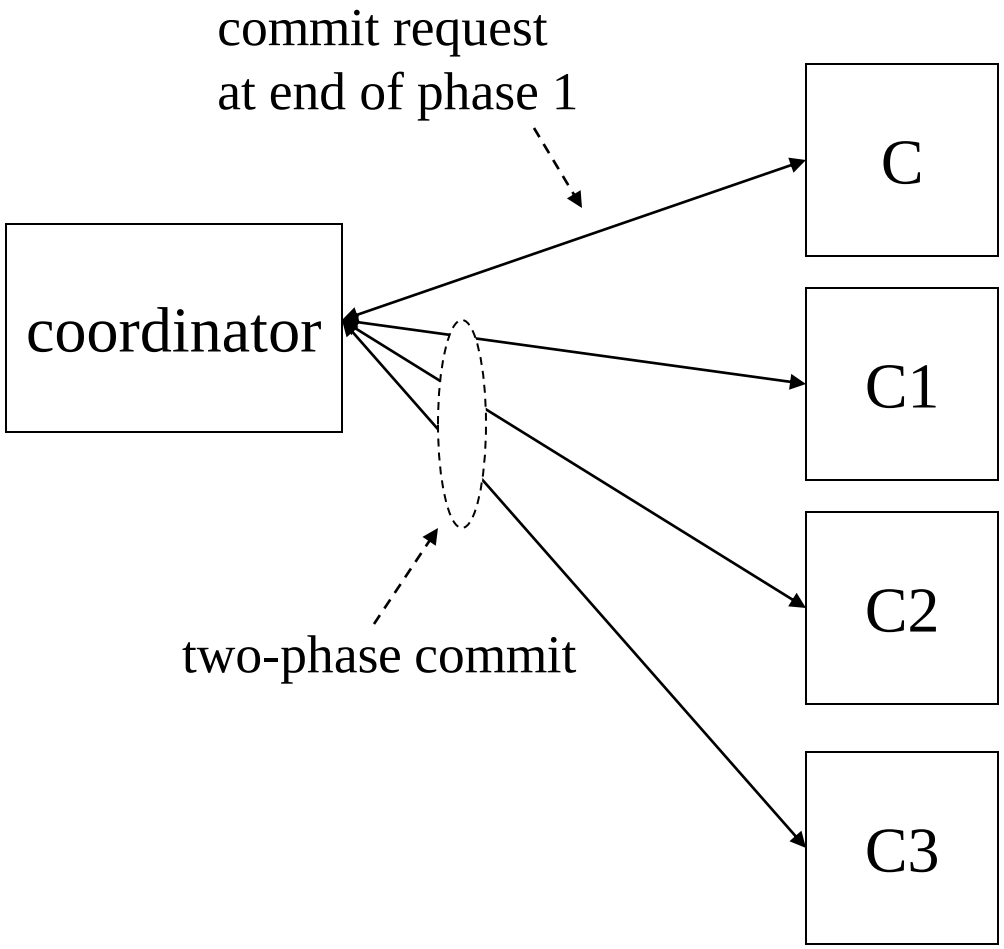
Linear Commit Protocol

- **Requires fewer messages than conventional two-phase commit. For n cohorts:**
  - **Linear commit requires 3(n - 1) messages**
  - **Two-phase commit requires 4n messages**
- **But:**
  - **Linear commit requires 3(n - 1) message times (messages are sent serially)**
  - **Two-phase commit requires 4 message times (messages are sent in parallel)**

Two-Phase Commit Without a Prepared State

- Assume exactly one cohort C, does not support a prepared state.
- Coordinator performs Phase 1 of two-phase commit protocol with all other cohorts
- If they all agree to commit, coordinator requests that C commit its subtransaction (in effect, requesting C to decide the transaction's outcome)
- C responds commit/abort, and the coordinator sends a commit/abort message to all other sites

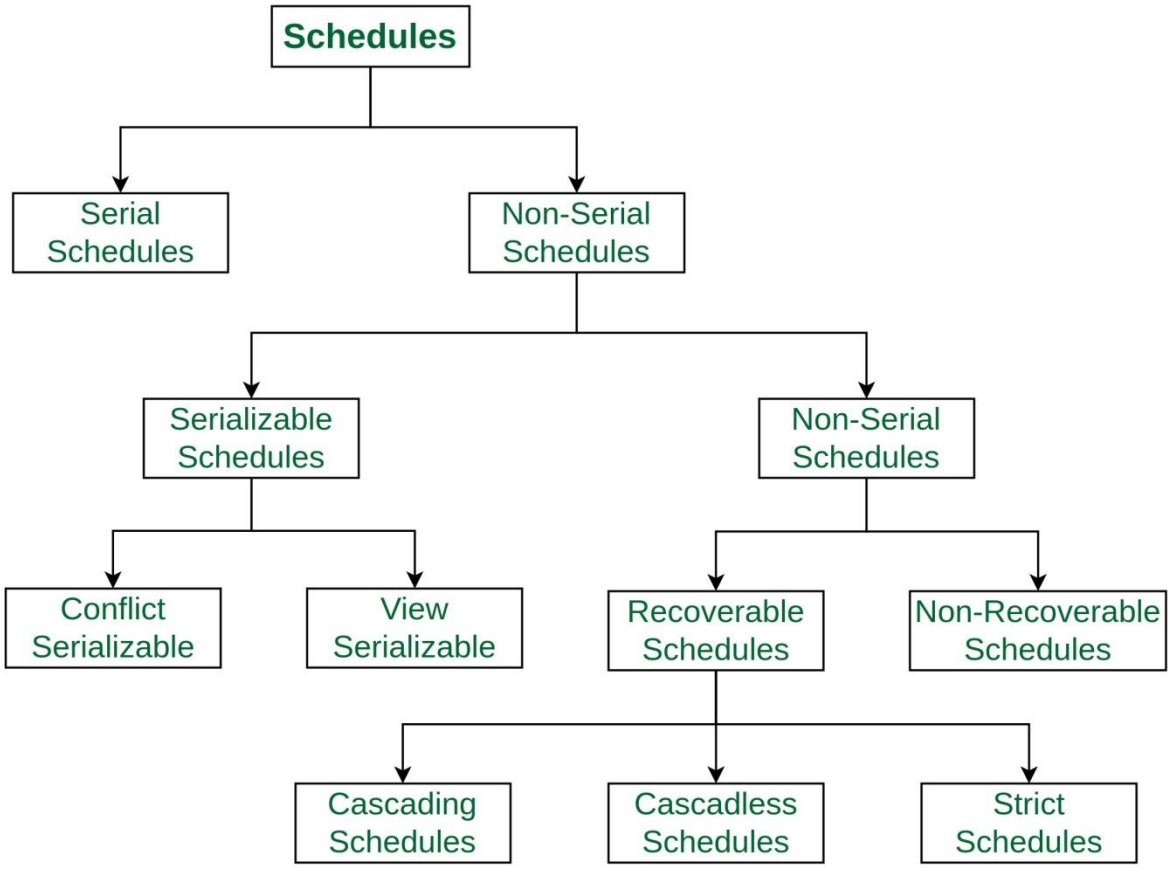
Two-Phase Commit Without a Prepared State



Serializability in distributed database

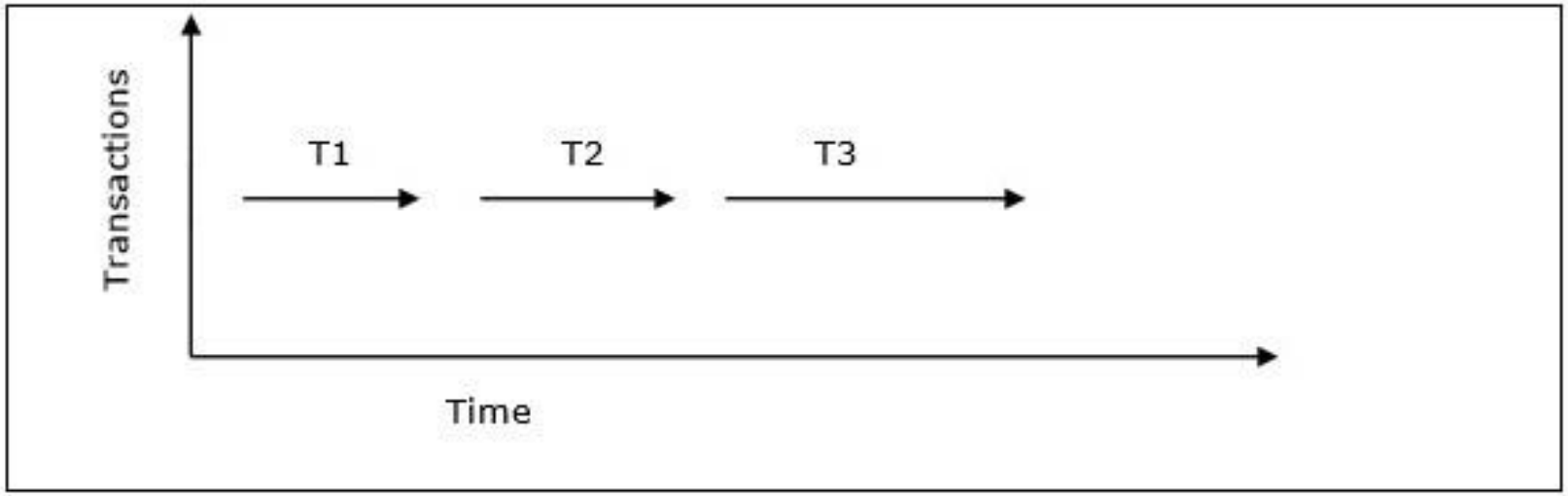
- In a system with a number of simultaneous transactions, a schedule is the total order of execution of operations.
- Given a schedule S comprising of n transactions, say T1, T2, T3…….Tn; for any transaction Ti, the operations in Ti must execute as laid down in the schedule S.

Types of schedules in DBMS



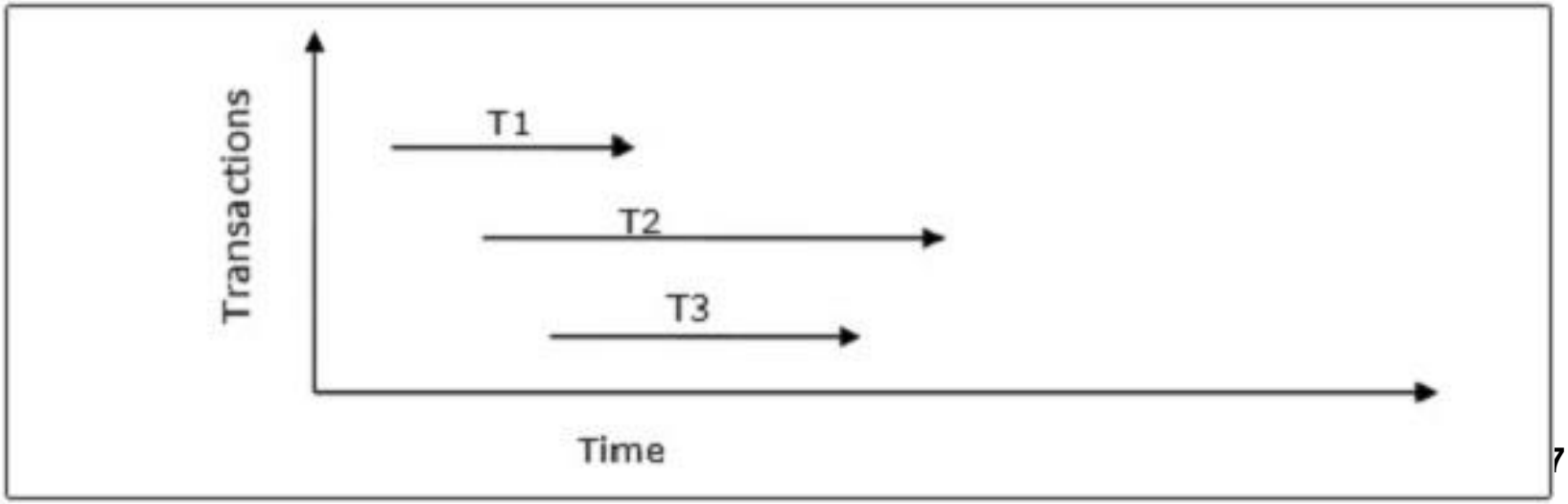
Serial Schedules

- In a serial schedule, at any point of time, only one transaction is active, i.e. there is no overlapping of transactions.
- This is depicted in the following graph



Parallel Schedules

- In parallel schedules, more than one transactions are active simultaneously, i.e. the transactions contain operations that overlap at time.
- This is depicted in the following graph



Serializable

- The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.
- **Serializable:**
  This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not.

Serializable

- On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete.
- The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions.
- Since concurrency is allowed in this case thus, multiple transactions can execute concurrently.
- A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:

    1. Conflict Serializable
    2. View Serializable

Conflict Serializable:

- A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.
- Two operations are said to be conflicting if all conditions satisfy:
  - They belong to different transactions
  - They operate on the same data item
  - At Least one of them is a write operation

View Serializable:

- A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions).
- A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

Non-Serializable:

- The non-serializable schedule is divided into two types,

1. Recoverable
2. Non-recoverable Schedule.

Recoverable Schedule:

- Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules.
- In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

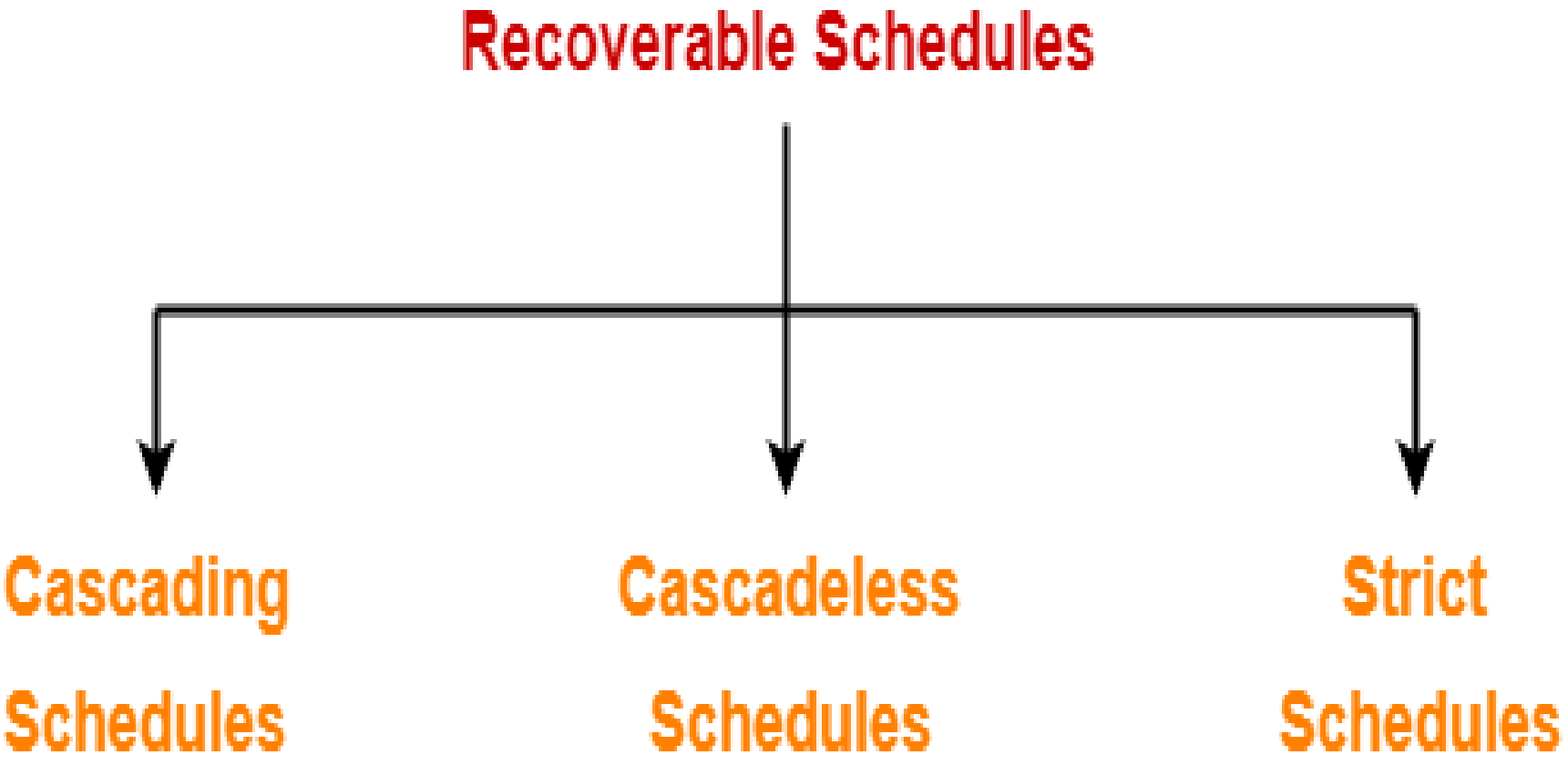
EXAMPLE FOR RECOVERABLE SCHEDULE

Example – Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.

Types of Recoverable Schedule



a. **Cascading Schedule:**

Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:

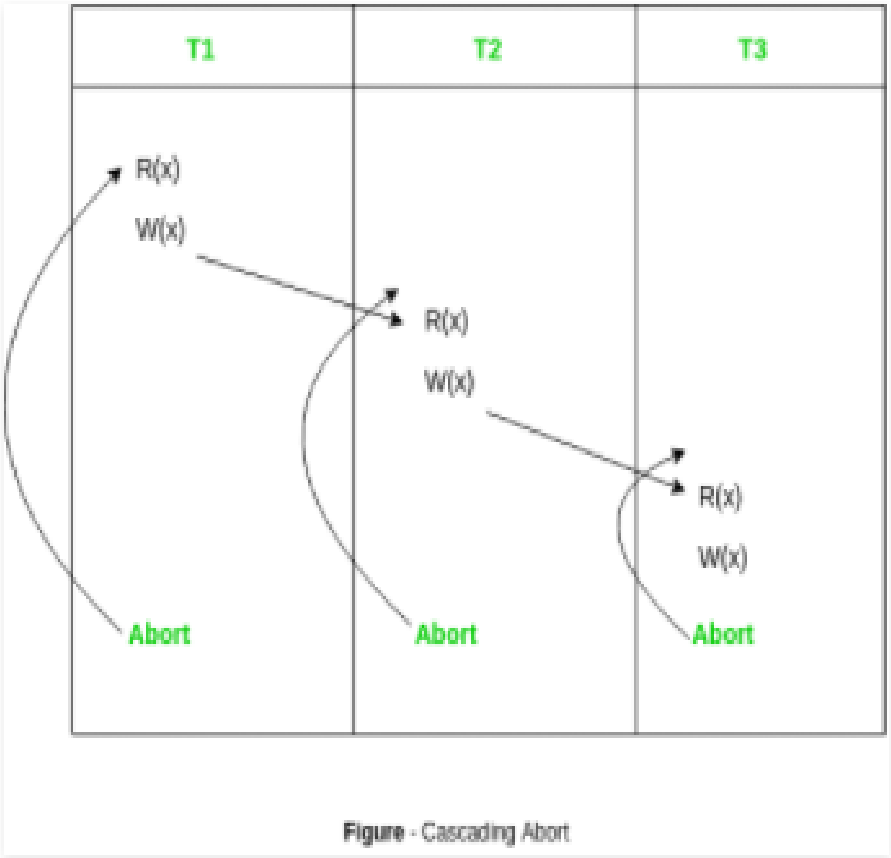


Figure - Cascading Abort

Cascadeless Schedule:

- Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules.
- Avoids that a single transaction abort leads to a series of transaction rollbacks.
- A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.
- In other words, if some transaction T_j wants to read value updated or written by some other transaction T_i, then the commit of T_j must read it after the commit of T_i.

Example for Cascadeless Schedule:

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
commit	
	R(A)
	commit

This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

Example for Cascadeless Schedule

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T_1 aborts, T_2 will have to be aborted too in order to maintain the correctness of the schedule as T_2 has already read the uncommitted value written by T_1.

Strict Schedule

- A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j.

- In other words, T_j can read or write updated or written value of T_i only after T_i commits/aborts.

Example for Strict Schedule

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

Non-Recoverable Schedule

Example: Consider the following schedule involving two transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

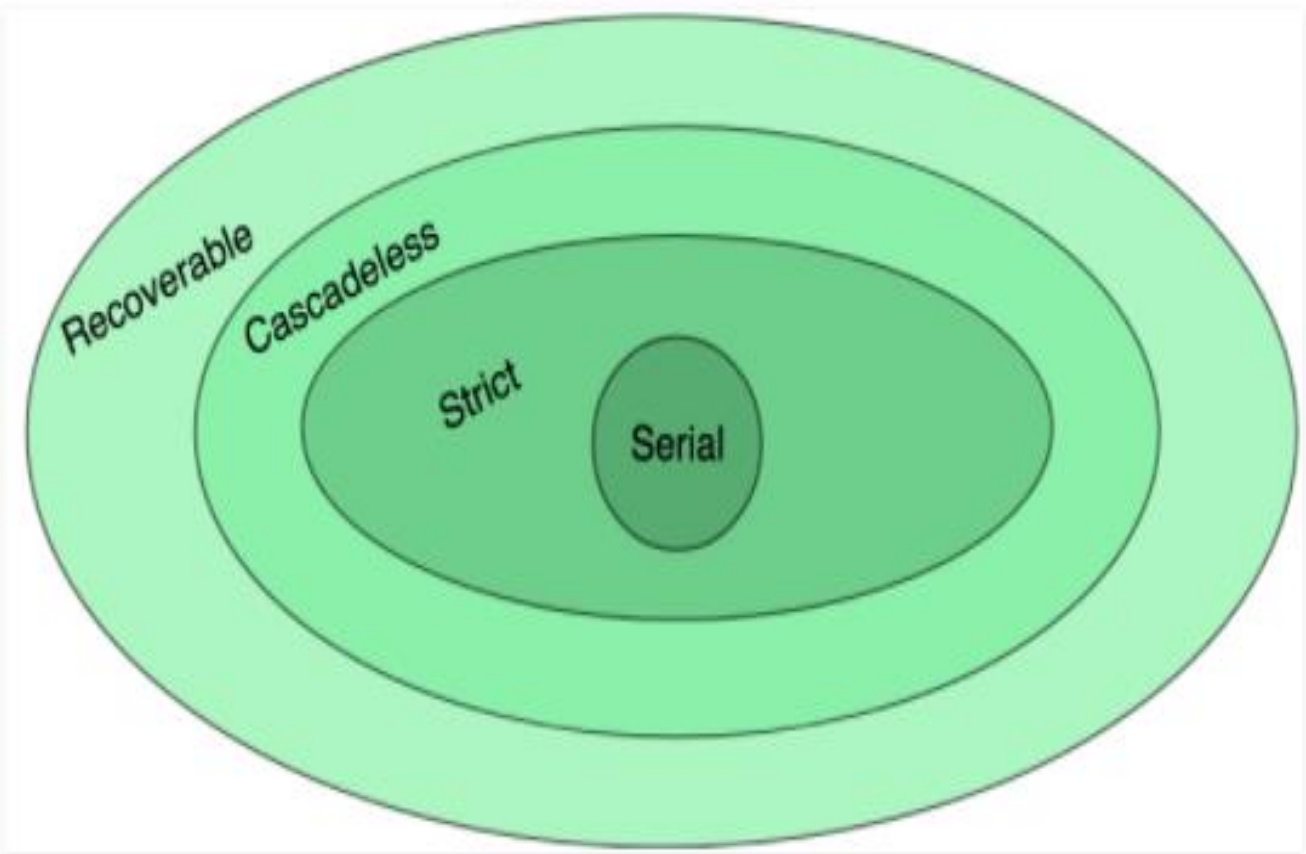
T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable**.

Note – It can be seen that:

- Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
- Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
- Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

Relationship Between Schedules

The relation between various types of schedules can be depicted as:



Global Deadlock

- **With distributed transaction:**
  - **A deadlock might not be detectable at any one site**
    - **Subtrans T_{1A} of T_1 at site A might wait for subtrans T_{2A} of T_2, while at site B, T_{2B} waits for T_{1B}**
  - **Since concurrent execution within a transaction is possible, a transaction might progress at some site even though deadlocked**
    - **T_{2A} and T_{1B} can continue to execute for a period of time**

Global Deadlock

- **Global deadlock cannot always be resolved by:**
  - **Aborting and restarting a single subtransaction, since data might have been communicated between cohorts**
  - **T_{2A}'s computation might depend on data received from T_{2B}. Restarting T_{2B} without restarting T_{2A} will not in general work.**

Global Deadlock Detection

- **Global deadlock detection is generally a simple extension of local deadlock detection**
  - **Check for a cycle when a cohort waits**
    - **If a cohort of T_1 is waiting for a cohort of T_2, coordinator of T_1 sends probe message to coordinator of T_2**
    - **If a cohort of T_2 is waiting for a cohort of T_3, coordinator of T_2 relays the probe to coordinator of T_3**
    - **If probe returns to coordinator of T_1 a deadlock exists**
  - **Abort a distributed transaction if the wait time of one of its cohorts exceeds some threshold**

Global Deadlock Prevention

- **Global deadlock prevention - use timestamps**
  - **For example an older transaction never waits for a younger one. The younger one is aborted.**

Global Isolation

- **If subtransactions at different sites run at different isolation levels, the isolation between concurrent distributed transactions cannot easily be characterized.**
- **Suppose all subtransactions run at SERIALIZABLE. Are distributed transactions as a whole serializable?**
  - **Not necessarily**
    - **T_{1A} and T_{2A} might conflict at site A, with T_{1A} preceding T_{2A}**
    - **T_{1B} and T_{2B} might conflict at site B, with T_{2B} preceding T_{1B}.**

Methods for Concurrency control

- There are main three methods for concurrency control. They are as follows:

1. Locking Methods

2. Time-stamp Methods

3. Optimistic Methods

Locking Methods of Concurrency Control

- A lock is a variable, associated with the data item, which controls the access of that data item.
- Locking is the most widely used form of the concurrency control.
- Locks are further divided into three fields:

1. Lock Granularity
2. Lock Types
3. Deadlocks

Lock Granularity

- A database is basically represented as a collection of named data items.
- The size of the data item chosen as the unit of protection by a concurrency control program is called GRANULARITY.
- Locking can take place at the following level :

1. Database level.
2. Table level.
3. Page level.
4. Row (Tuple) level.
5. Attributes (fields) level.

Level of locking

i. Database level Locking :

- At database level locking, the entire database is locked.
- Thus, it prevents the use of any tables in the database by transaction T2 while transaction T1 is being executed.
- Database level of locking is suitable for batch processes.
- Being very slow, it is unsuitable for on-line multi-user DBMSs.

Level of locking

ii. Table level Locking :

- At table level locking, the entire table is locked.
- Thus, it prevents the access to any row (tuple) by transaction T2 while transaction T1 is using the table.
- If a transaction requires access to several tables, each table may be locked.
- However, two transactions can access the same database as long as they access different tables. Table level locking is less restrictive than database level.
- Table level locks are not suitable for multi-user DBMS

Level of locking

iii. Page level Locking :

- At page level locking, the entire disk-page (or disk-block) is locked.
- A page has a fixed size such as 4 K, 8 K, 16 K, 32 K and so on.
- A table can span several pages, and a page can contain several rows (tuples) of one or more tables.
- Page level of locking is most suitable for multi-user DBMSs.

Level of locking

iv. Row (Tuple) level Locking :

- At row level locking, particular row (or tuple) is locked.
- A lock exists for each row in each table of the database.
- The DBMS allows concurrent transactions to access different rows of the same table, even if the rows are located on the same page.
- The row level lock is much less restrictive than database level, table level, or page level locks.
- The row level locking improves the availability of data.
- However, the management of row level locking requires high overhead cost.

Level of locking

v. Attributes (fields) level Locking :

- At attribute level locking, particular attribute (or field) is locked.
- Attribute level locking allows concurrent transactions to access the same row, as long as they require the use of different attributes within the row.
- The attribute level lock yields the most flexible multi-user data access.
- It requires a high level of computer overhead.

Lock Types

The DBMS mainly uses following types of locking techniques.

- Binary Locking
- Shared / Exclusive Locking
- Two - Phase Locking (2PL)

Binary Locking

- A binary lock can have two states or values: locked and unlocked (or 1 and 0, for simplicity).
- A distinct lock is associated with each database item X.
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item.
- If the value of the lock on X is 0, the item can be accessed when requested.
- We refer to the current value (or state) of the lock associated with item X as LOCK(X).

Operations of Binary Locking

Two operations, lock_item and unlock_item, are used with binary locking.

Lock_item(X):

A transaction requests access to an item X by first issuing a lock_item(X) operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction locks the item) and the transaction is allowed to access item X.

Unlock_item (X):

When the transaction is through using the item, it issues an unlock_item(X) operation, which sets LOCK(X) to 0 (unlocks the item) so that X may be accessed by other transactions. Hence, a binary lock enforces mutual exclusion on the data item ; i.e., at a time only one transaction can hold a lock.

Shared / Exclusive Locking

Shared lock :

- These locks are referred as read locks, and denoted by 'S'.
- If a transaction T has obtained Shared-lock on data item X, then T can read X, but cannot write X.
- Multiple Shared lock can be placed simultaneously on a data item.

Exclusive lock :

- These Locks are referred as Write locks, and denoted by 'X'
- If a transaction T has obtained Exclusive lock on data item X, then T can be read as well as write X.
- Only one Exclusive lock can be placed on a data item at a time.
- This means multiple transactions does not modify the same data simultaneously.

Two-Phase Locking (2PL)

- Two-phase locking (also called 2PL) is a method or a protocol of controlling concurrent processing in which all locking operations precede the first unlocking operation.
- Thus, a transaction is said to follow the two-phase locking protocol if all locking operations (such as read_Lock, write_Lock) precede the first unlock operation in the transaction.
- Two-phase locking is the standard protocol used to maintain level 3 consistency 2PL defines how transactions acquire and relinquish locks.

Two-Phase Locking (2PL)

- The essential discipline is that after a transaction has released a lock it may not obtain any further locks.
- 2PL has the following two phases:

A **growing** phase, in which a transaction acquires all the required locks without unlocking any data. Once all locks have been acquired, the transaction is in its locked point.

A **shrinking** phase, in which a transaction releases all locks and cannot obtain any new lock.

Two-Phase Locking (2PL)

Example:

- A deadlock exists two transactions A and B exist in the following example:

Transaction A = access data items X and Y
Transaction B = access data items Y and X

- Here, Transaction-A has acquired lock on X and is waiting to acquire lock on y.
- While, Transaction-B has acquired lock on Y and is waiting to acquire lock on X.
- But, none of them can execute further.

Deadlocks

- A deadlock is a condition in which two (or more) transactions in a set are waiting simultaneously for locks held by some other transaction in the set.
- Neither transaction can continue because each transaction in the set is on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.
- Thus, a deadlock is an impasse that may result when two or more transactions are each waiting for locks to be released that are held by the other.

Deadlocks

- Transactions whose lock requests have been refused are queued until the lock can be granted.
- A deadlock is also called a circular waiting condition where two transactions are waiting (directly or indirectly) for each other.
- Thus in a deadlock, two transactions are mutually excluded from accessing the next record required to complete their transactions, also called a deadly embrace.

Deadlock Detection and Prevention:

Deadlock detection:

- This technique allows deadlock to occur, but then, it detects it and solves it.
- Here, a database is periodically checked for deadlocks.
- If a deadlock is detected, one of the transactions, involved in deadlock cycle, is aborted. other transaction continue their execution.
- An aborted transaction is rolled back and restarted.

Deadlock Prevention

- Deadlock prevention technique avoids the conditions that lead to deadlocking.
- It requires that every transaction lock all data items it needs in advance.
- If any of the items cannot be obtained, none of the items are locked.
- In other words, a transaction requesting a new lock is aborted if there is the possibility that a deadlock can occur.

Deadlock Prevention

- Thus, a timeout may be used to abort transactions that have been idle for too long.
- This is a simple but indiscriminate approach.
- If the transaction is aborted, all the changes made by this transaction are rolled back and all locks obtained by the transaction are released.
- The transaction is then rescheduled for execution.
- Deadlock prevention technique is used in two-phase locking.

Time-Stamp Methods for Concurrency control

- Timestamp is a unique identifier created by the DBMS to identify the relative starting time of a transaction.
- Typically, timestamp values are assigned in the order in which the transactions are submitted to the system.
- So, a timestamp can be thought of as the transaction start time.
- Therefore, time stamping is a method of concurrency control in which each transaction is assigned a transaction timestamp.

Time-Stamp Methods for Concurrency control

Timestamps must have two properties namely

1. Uniqueness : The uniqueness property assures that no equal timestamp values can exist.
2. monotonicity : monotonicity assures that timestamp values always increase.

Timestamp are divided into further fields :

1. Granule Timestamps
2. Timestamp Ordering
3. Conflict Resolution in Timestamps

Granule Timestamps

- Granule timestamp is a record of the timestamp of the last transaction to access it.
- Each granule accessed by an active transaction must have a granule timestamp.
- A separate record of last Read and Write accesses may be kept.
- Granule timestamp may cause.
- Additional Write operations for Read accesses if they are stored with the granules.
- The problem can be avoided by maintaining granule timestamps as an in-memory table.

Granule Timestamps

- The table may be of limited size, since conflicts may only occur between current transactions.
- An entry in a granule timestamp table consists of the granule identifier and the transaction timestamp.
- The record containing the largest (latest) granule timestamp removed from the table is also maintained.
- A search for a granule timestamp, using the granule identifier, will either be successful or will use the largest removed timestamp.

Timestamp Ordering

Following are the three basic variants of timestamp-based methods of concurrency control:

1. Total timestamp ordering
2. Partial timestamp ordering
3. Multiversion timestamp ordering

- The total timestamp ordering algorithm depends on maintaining access to granules in timestamp order by aborting one of the transactions involved in any conflicting access.
- No distinction is made between Read and Write access, so only a single value is required for each granule timestamp .

Partial timestamp ordering

- In a partial timestamp ordering, only non-permutable actions are ordered to improve upon the total timestamp ordering.
- In this case, both Read and Write granule timestamps are stored.
- The algorithm allows the granule to be read by any transaction younger than the last transaction that updated the granule.
- A transaction is aborted if it tries to update a granule that has previously been accessed by a younger transaction.
- The partial timestamp ordering algorithm aborts fewer transactions than the total timestamp ordering algorithm, at the cost of extra storage for granule timestamps

Multiversion Timestamp ordering

- The multiversion timestamp ordering algorithm stores several versions of an updated granule, allowing transactions to see a consistent set of versions for all granules it accesses.
- So, it reduces the conflicts that result in transaction restarts to those where there is a Write-Write conflict.
- Each update of a granule creates a new version, with an associated granule timestamp.
- A transaction that requires read access to the granule sees the youngest version that is older than the transaction.
- That is, the version having a timestamp equal to or immediately below the transaction's timestamp.

Conflict Resolution in Timestamps

- To deal with conflicts in timestamp algorithms, some transactions involved in conflicts are made to wait and to abort others.
- Following are the main strategies of conflict resolution in timestamps:

➢ **WAIT-DIE:**

- The older transaction waits for the younger if the younger has accessed the granule first.
- The younger transaction is aborted (dies) and restarted if it tries to access a granule after an older concurrent transaction.

Conflict Resolution in Timestamps

➢ **WOUND-WAIT:**

- The older transaction pre-empts the younger by suspending (wounding) it if the younger transaction tries to access a granule after an older concurrent transaction.
- An older transaction will wait for a younger one to commit if the younger has accessed a granule that both want.

Conflict Resolution in Timestamps

- The handling of aborted transactions is an important aspect of conflict resolution algorithm.
- In the case that the aborted transaction is the one requesting access, the transaction must be restarted with a new (younger) timestamp.
- It is possible that the transaction can be repeatedly aborted if there are conflicts with other transactions.
- An aborted transaction that had prior access to granule where conflict occurred can be restarted with the same timestamp.
- This will take priority by eliminating the possibility of transaction being continuously locked out.

Drawbacks of Time-stamp

- Each value stored in the database requires two additional timestamp fields, one for the last time the field (attribute) was read and one for the last update.
- It increases the memory requirements and the processing overhead of database

Optimistic Methods of Concurrency Control

- The optimistic method of concurrency control is based on the assumption that conflicts of database operations are rare and that it is better to let transactions run to completion and only check for conflicts before they commit.
- An optimistic concurrency control method is also known as validation or certification methods.
- No checking is done while the transaction is executing.

Optimistic Methods of Concurrency Control

- The optimistic method does not require locking or time stamping techniques.
- Instead, a transaction is executed without restrictions until it is committed.
- In optimistic methods, each transaction moves through the following phases:

1. Read phase.
2. Validation or certification phase.
3. Write phase.

Read phase

- In a Read phase, the updates are prepared using private (or local) copies (or versions) of the granule.
- In this phase, the transaction reads values of committed data from the database, executes the needed computations, and makes the updates to a private copy of the database values.
- All update operations of the transaction are recorded in a temporary update file, which is not accessed by the remaining transactions.
- It is conventional to allocate a timestamp to each transaction at the end of its Read to determine the set of transactions that must be examined by the validation procedure.
- These set of transactions are those who have finished their Read phases since the start of the transaction being verified

Validation or certification phase

- In a validation (or certification) phase, the transaction is validated to assure that the changes made will not affect the integrity and consistency of the database.
- If the validation test is positive, the transaction goes to the write phase.
- If the validation test is negative, the transaction is restarted, and the changes are discarded.
- Thus, in this phase the list of granules is checked for conflicts.
- If conflicts are detected in this phase, the transaction is aborted and restarted.

Validation or certification phase

- The validation algorithm must check that the transaction has :
  - Seen all modifications of transactions committed after it starts.
  - Not read granules updated by a transaction committed after its start.

Write phase

- In a Write phase, the changes are permanently applied to the database and the updated granules are made public.
- Otherwise, the updates are discarded and the transaction is restarted.
- This phase is only for the Read-Write transactions and not for Read-only transactions.

Advantages of Optimistic Methods for Concurrency Control :

- This technique is very efficient when conflicts are rare. The occasional conflicts result in the transaction roll back.
- The rollback involves only the local copy of data, the database is not involved and thus there will not be any cascading rollbacks.

Problems of Optimistic Methods for Concurrency Control :

- Conflicts are expensive to deal with, since the conflicting transaction must be rolled back.
- Longer transactions are more likely to have conflicts and may be repeatedly rolled back because of conflicts with short transactions.

Applications of Optimistic Methods for Concurrency Control :

- Only suitable for environments where there are few conflicts and no long transactions.
- Acceptable for mostly Read or Query database systems that require very few update transactions