Chapter 10
Transaction Manager Concepts

Mijin An
meeeeeejin@gmail.com

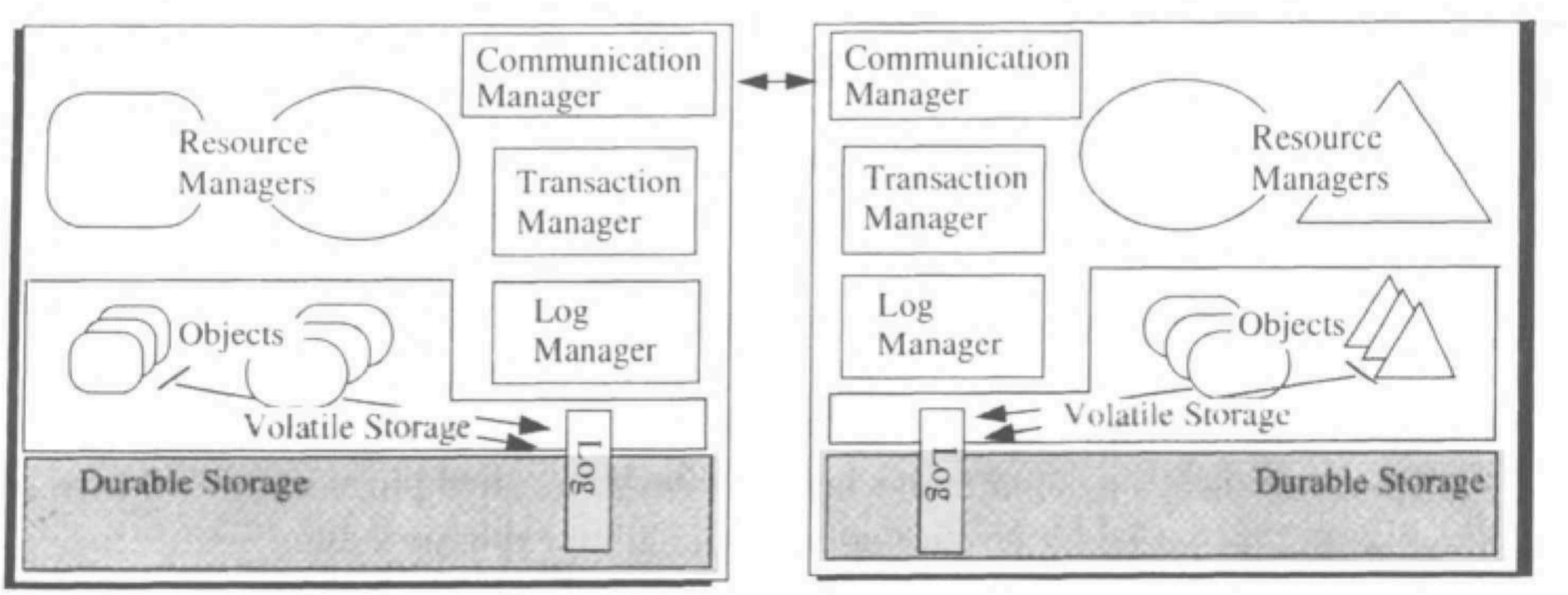
Overview: Transaction Manager

- The transaction manager simply **gathers information that will be needed in case of failure**
  - It does very little during normal processing

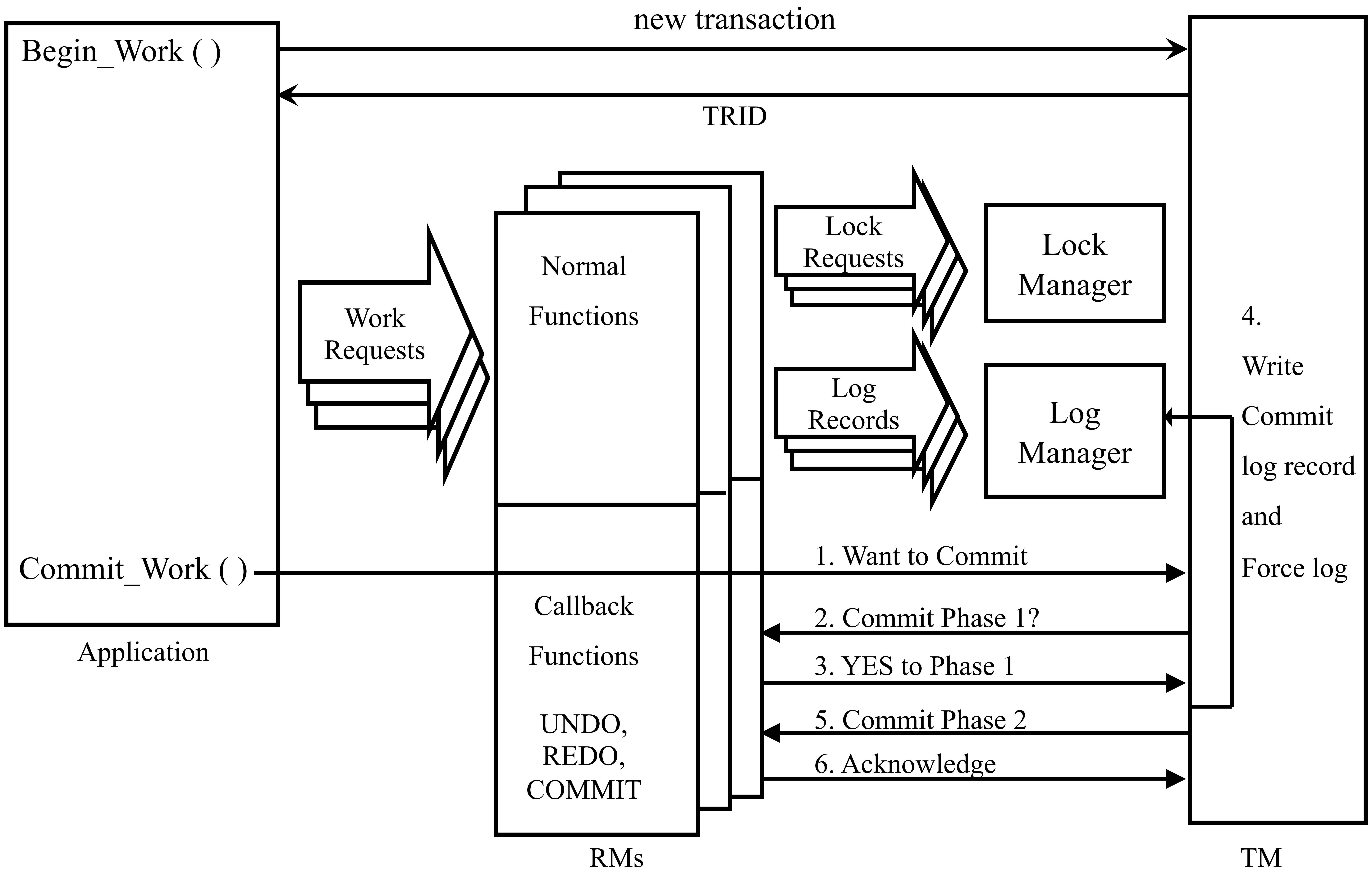
- The transaction manager furnishes the **A, C, D** of ACID
  - **Atomicity**: Undoing aborted transaction and redoing committed ones
  - **Consistency**: Aborting any transactions that fail to pass the RM consistency tests at commit
  - **Durability**: Forcing all log records of committed transactions to durable memory as part of commit processing and redoing any recently committed work at restart

Transaction Manager Interfaces

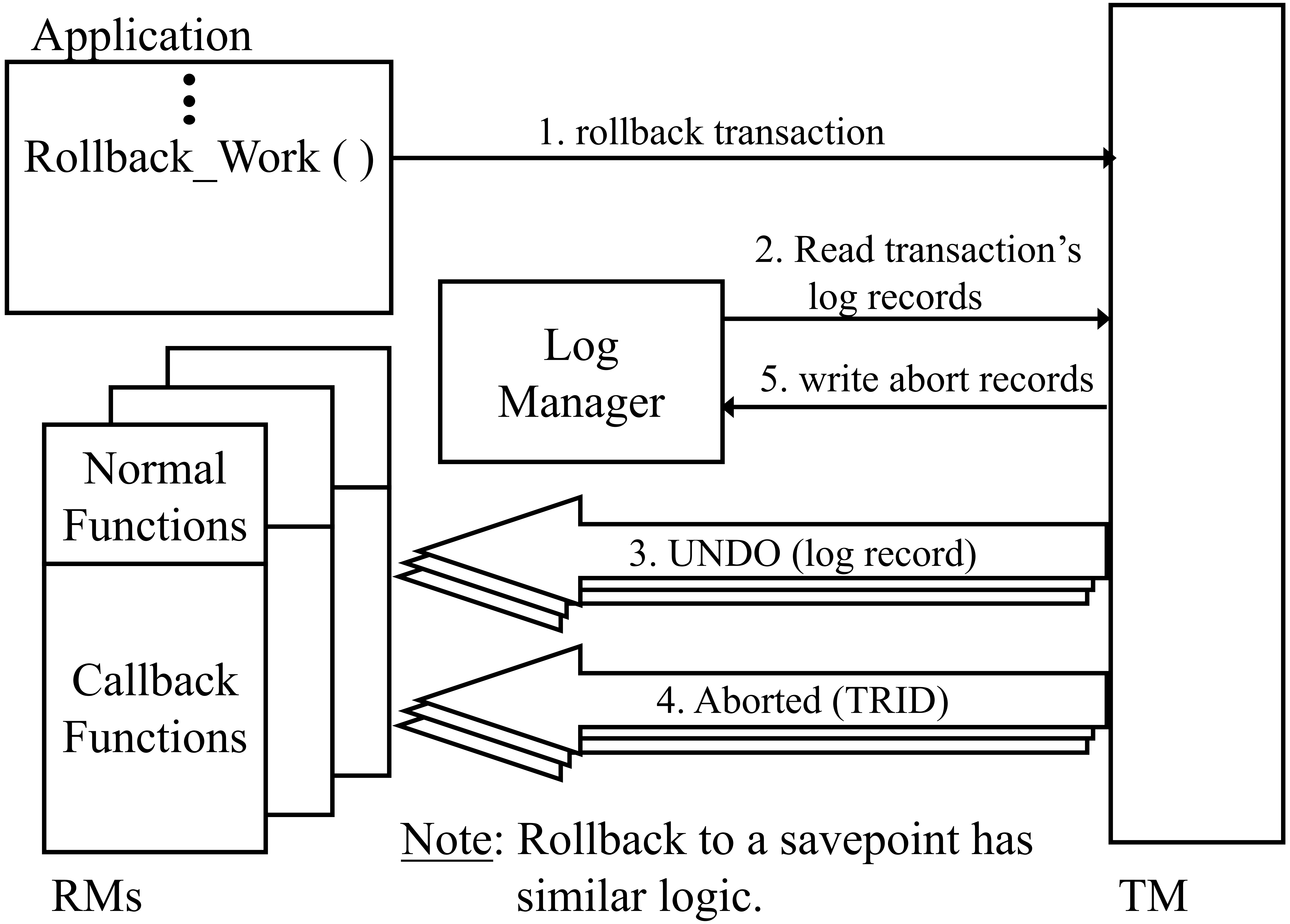
- The transaction manager with the log and lock manager supplies the mechanism to build RMs and computations with the ACID properties



Normal Execution



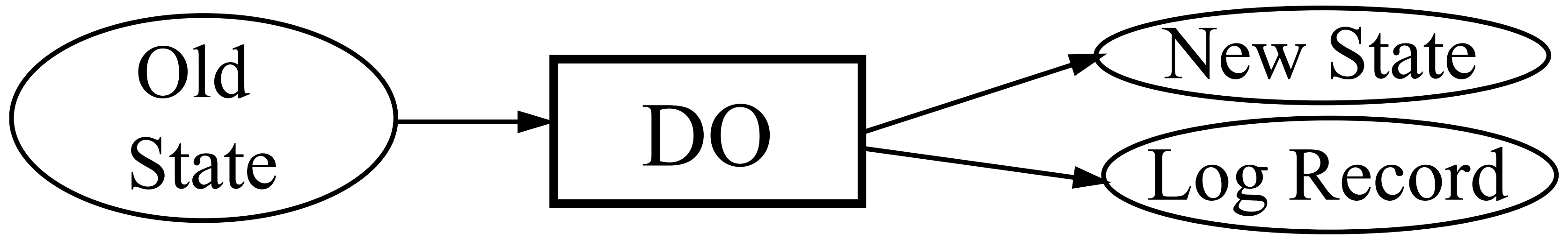
Transaction Abort



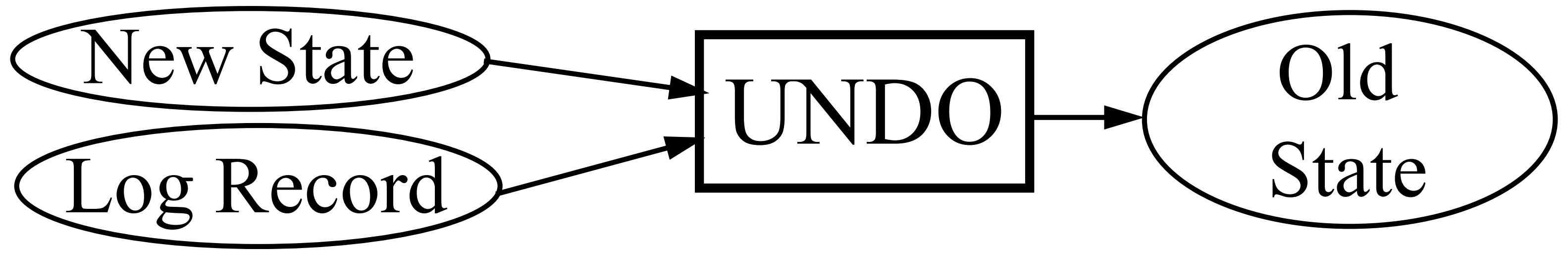
Note: Rollback to a savepoint has similar logic.

The DO-UNDO-REDO Protocol

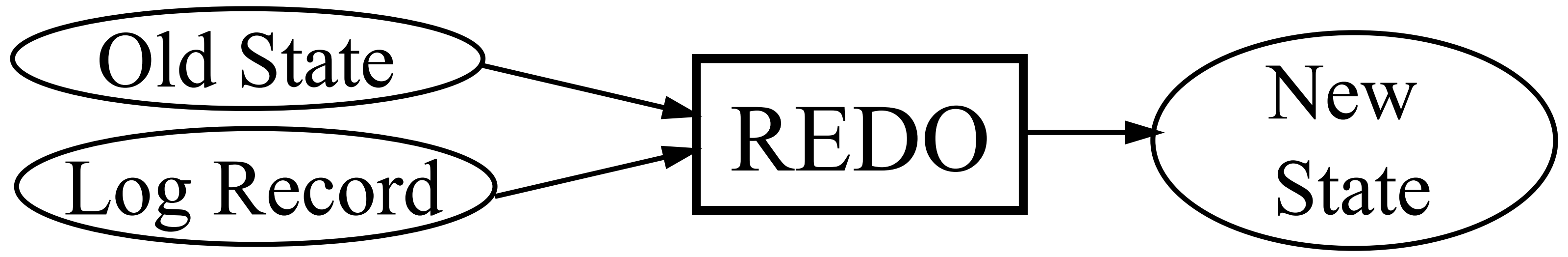
- The DO-UNDO-REDO protocol is a programming style for RMs implementing transactional objects
  - **DO** program:



  - **UNDO** program:



  - **REDO** program:



- RMs have following structure:

RM {
Normal Function: **DO** program
Callback Functions: **UNDO & REDO** program

Restart Process

1. The transaction manager (TM) regularly invokes checkpoints during normal processing
    - It informs each RM to checkpoint its state to persistent memory

2. At restart, the TM scans the log table forward from the most recent checkpoint to the end

3. For each log record, the TM calls the REDO() callback of the RM

4. For each transaction that has not committed, the TM calls the UNDO() callback of the RM to undo it to the most recent persistent savepoint

Value Logging

- **Each log record contains the old and new object states**
- It is also called **old value-new value logging** or **physical logging** because it records the physical addresses and values of objects

```
struct value_log_record_for_page_update {
    int      opcode;                /* opcode will say page update */
    filename fname;                 /* name of file that was updated */
    long     pageno;                /* page that was updated */
    char     old_value[PAGESIZE];   /* old value of page */
    char     new_value[PAGESIZE];   /* new value of page */
}
```

- If the object state is small, it is a good design
- UNDO and REDO are idempotent

Logical Logging

- **Each log record contains the name of a operation and its parameters, rather than the object values themselves; For example:**

```
<insert op, table name = A, record value = r>
```

- It takes the DO-UNDO-REDO model literally
- It assumes that **each action is atomic** and that in each failure situation the system state will be **action consistent**:
  - Each logical action will have been completely done or completely undone

- <u>Problem</u>: Partially complete actions can fail; And the UNDO of these partial actions will not be presented with an action-consistent state

Physiological Logging (1)

- Physiological logging is a compromise between logical and physical logging
  - **Physical-to-a-page, logical-within-a-page**

```
<insert op, base filename = A, page number = 508, record value = r>
<insert op, index1 filename = B, page number = 72, index1 record value = keyB of r = s>
<insert op, index2 filename = C, page number = 94, index2 record value = keyC of r = t>
```

- **Fundamental idea**:
  - Log records are generated on a per-page basis
  - Log records are designed to make logical transformation of pages

Physiological Logging (2)

- There are the ideas that motivate physiological logging:
  - **Page actions**
    - Complex actions can be structured as a sequence of page actions
  - **Mini-transaction**
    - Page actions can be structured as mini-transactions that use logical logging
      - When the action completes, the object is updated
      - An UNDO-REDO log record is created to cover that action
  - **Log-object consistency**
    - It is possible to structure the system so that at restart, the persistent state is page-action consistent

Physiological Logging During Online Operation

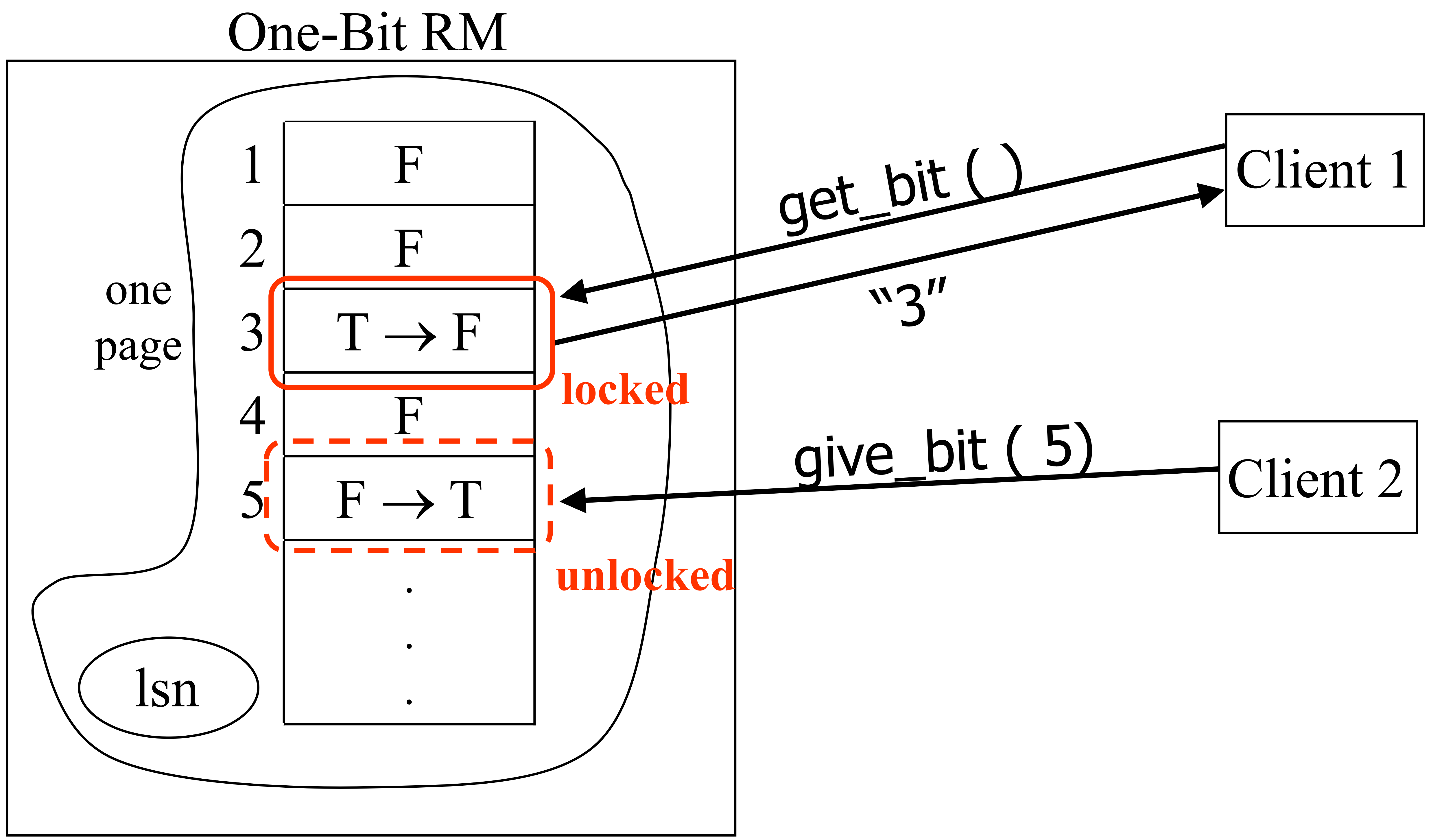
- We call **normal operations without failures** as online operations
- To allow updates, all page changes must be structured as **mini-transactions**:

```
Mini_trans()
    Lock the object in exclusive mode
    Transform the object
    Generate an UNDO–REDO log record
    Unlock the object
```

- The mini-transaction approach ensures online consistency:
  - **Page-action consistency**
    - Volatile and persistent memory are in a page-consistent state, and each page reflects the most recent updates to it
  - **Log consistency**
    - The log contains a history of all update to pages

The One-Bit Resource Manager

- This RM manages an array of bits stored in a single page
- Each bit is either free (TRUE) or busy (FALSE)



Meaning of Page and Log Consistency

- **Page consistency**
  - No clean free bit has been given to any transaction
  - Every clean busy bit has been given to exactly one transaction
  - Dirty bits are locked in exclusive mode by the transactions that modified them
  - The log sequence number (LSN) reflects the most recent log record for this page

- **Log consistency**
  - The log contains a log record for every completed mini-transaction update to the page

The FIX Rule

- While the semaphore is set, the page is said to be fixed, and releasing the page is unfixing it

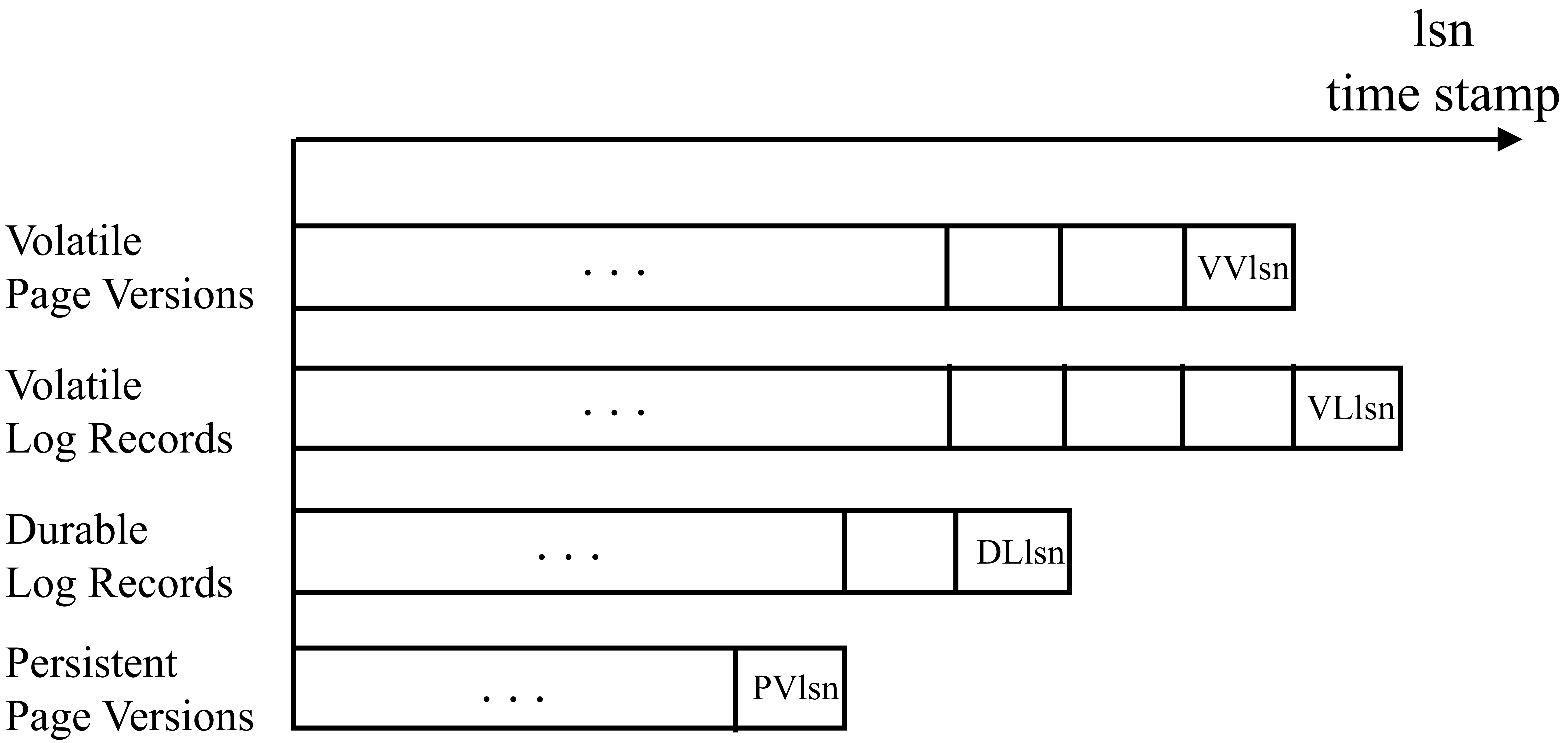
- **FIX Rule**:
  - Get the page semaphore in exclusive mode prior to altering the page
  - Get the semaphore in shared or exclusive mode prior to reading the page
  - Hold the semaphore until the page and log are again consistent, and the read or update is complete

- This is just two-phase locking at the page-semaphore level

Online Consistency & Restart Consistency (1)

- **Online log consistency** requires that volatile log contain all log records up to and including vvlsn

`VVlsn ≤ VLlsn`



Online Consistency & Restart Consistency (2)

- **Restart log consistency** ensures that if a transaction has committed with commit_lsn, then that commit record is in the durable log

$$\texttt{commit\_lsn} \le \texttt{DLlsn}$$

- Restart log consistency guarantees that if version X of the volatile copy overwrites the durable copy, then the log records for version X are already present in the durable log

$$\texttt{VVlsn} \le \texttt{DLlsn}$$

- At restart, all volatile memory is reset and must be reconstructed from persistent memory

$$\texttt{PVlsn} \le \texttt{DLlsn}$$
$$\texttt{commit\_lsn} \le \texttt{DLlsn}$$

Write-Ahead Log (WAL) Protocol

- Each volatile page has an LSN field and each update must maintain the page LSN field

- When a page is about to be copied to persistent memory, the copier must first use the log manager to copy all log records up to and including the page's LSN to durable memory

- Once the force completes, the volatile version of the page can overwrite the persistent version of the page

- The page must be fixed during the writes and during the copies, to guarantee page-action consistency

Force-Log-at-Commit

- **Question**: What if no pages were copied to persistent memory, and the transaction committed?
  - If the system were to restart immediately, there would be no record of the transaction's updates, and the transaction could not be redone
  - This would violate **transaction durability**

- **Solution**: **Force-Log-at-Comimt**
  - The transaction's log records must be moved to durable memory as part of commit
  - When a transaction commits, the TM writes a commit log record and requests the log manager to flush the log

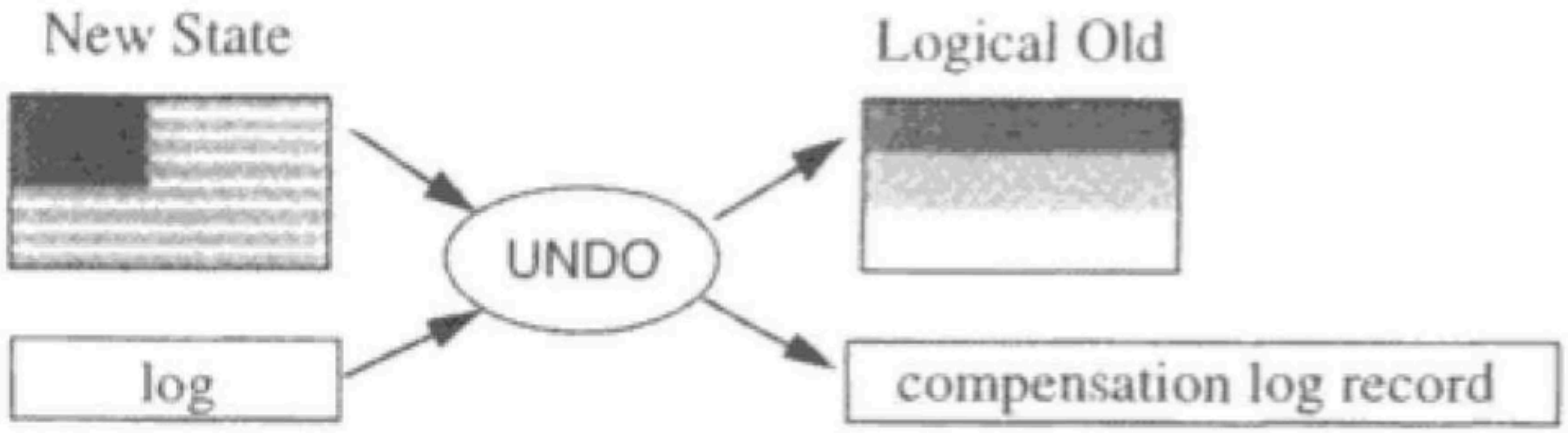
Summary of Physiological Logging

- The physiological logging assumptions of page-action consistency and log consistency require the RM to observe the following three rules:
  - **Fix rule**
    - Cover all page reads and writes with the page semaphore
  - **Write-ahead log (WAL)**
    - Force the page's log records prior to overwriting its persistent copy
  - **Force-log-at-commit**
    - Force the transaction's log records as part of commit

UNDO: Compensation Log Records

- **Question**: What should the page LSN become when an action is undone?
  - If subsequent updates to the page by other transactions have advanced the LSN, the LSN should not be set back to its original value

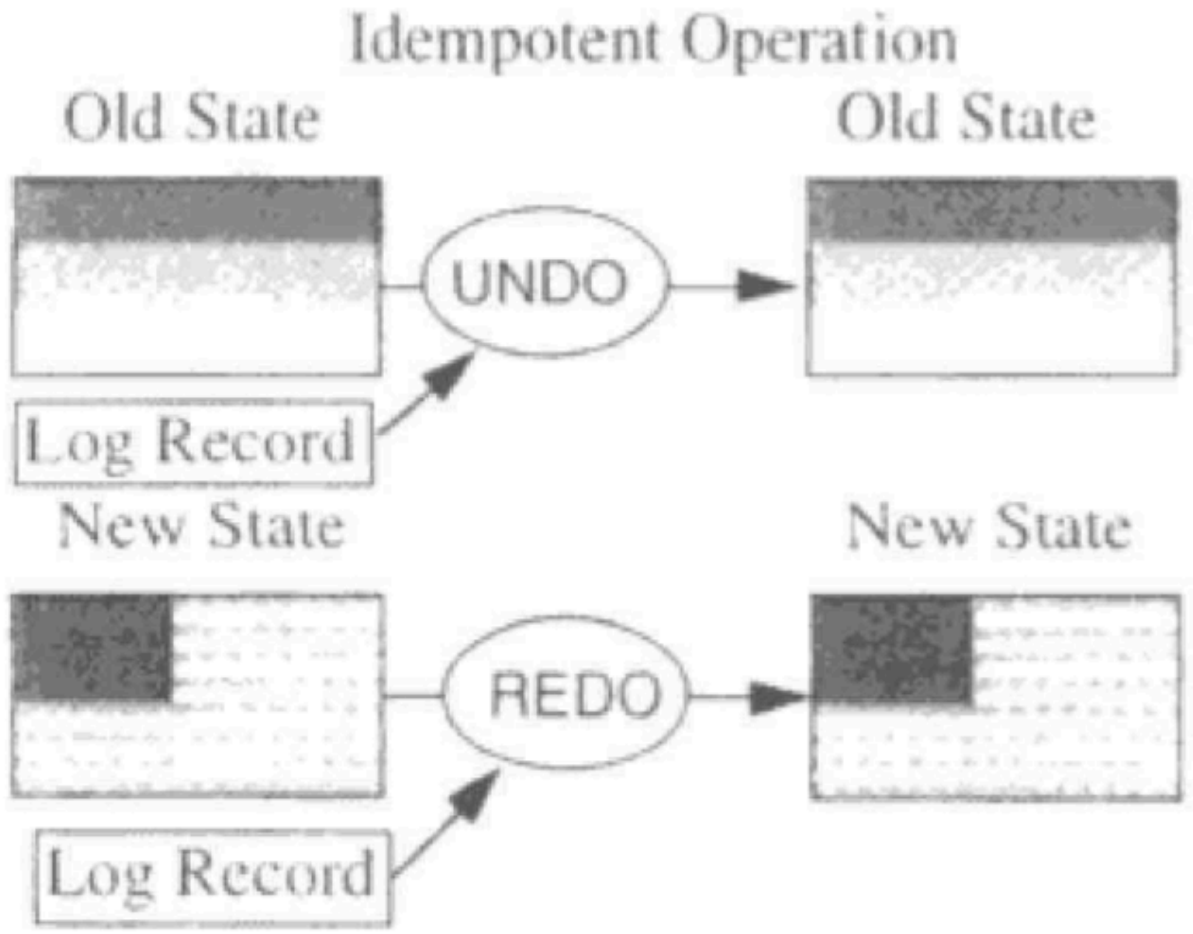
- **Strategy**:
  - UNDO looks just like a new action that generates a new log record — called a **compensation log record**
  - This approach makes page LSNs monotonic

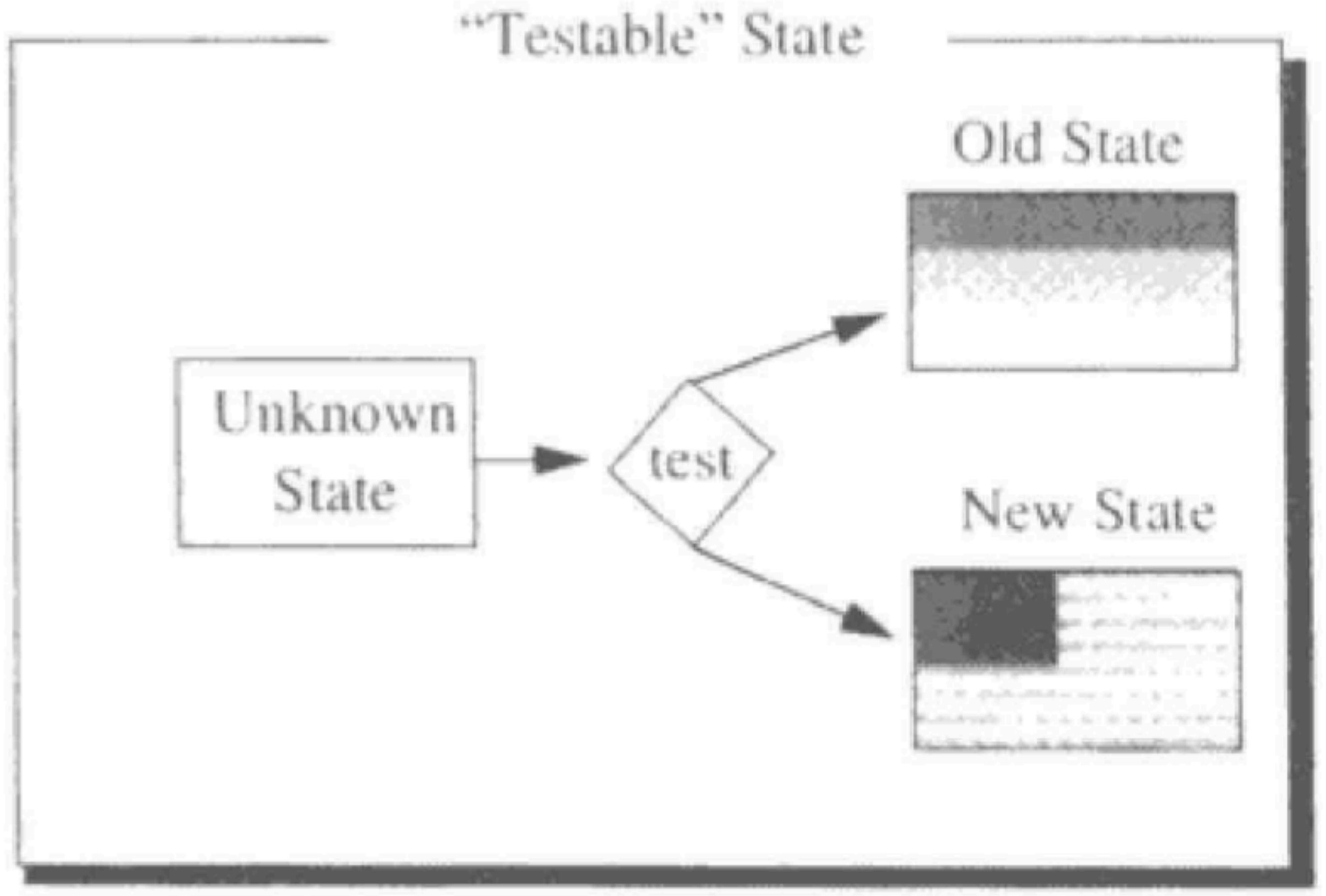


Idempotence & Testable

- **Idempotent** operation:
  - If the UNDO or REDO operation can be repeated an arbitrary number of times and still result in the correct state, the operation is idempotent

- **Testable** state:
  - If the old and new states can be discriminated by the system, the state is testable





Idempotence of Physiological REDO (1)

- Repeated REDOs can arise from repeated failures during restart

- **Example**: Suppose the following physiological log record were redone many times:

```
<insert op, base filename, page number, record value>
```

  - If no special care were taken, this repeated REDO would result in many inserts of the record into the page

Idempotence of Physiological REDO (2)

- **Solution**:

```
idempotent_physiologic_redo (page, logrec)
{
  if (page_lsn < logrec_lsn)
    redo(page, logrec);
}
```

  - This logic makes **physiological REDOs idempotent**

Two-Phase Commit: Making Computations Atomic

- When a transaction is about to commit, each participant in the transaction is given a chance to vote on whether the transaction is a consistent state transformation
- If all the RMs vote yes, the transaction can **commit**
- If any vote no, the transaction is **aborted**
- This is the **two-phase commit** protocol
  - Phase 1: Voting phase
  - Phase 2: Actual commit phase

Two-Phase Commit: Commit

- **Phase 1:**
  - **Prepare**: Invoke each RM asking for its vote
  - **Decide**: If all vote yes, durably write the transaction commit log record
- **Phase 2:**
  - **Commit**: Invoke each RM telling it the commit decision
  - **Complete**: When all acknowledge the commit message, write a commit completion record to the log; When the completion message is durable, deallocate the live transaction state

Two-Phase Commit: Abort

- If any RM votes **no** during the prepare step, or if it doest **not respond** at all, then the transaction cannot commit

- The simplest thing to do in this case is to **roll back** the transaction by calling `Abort_Work()`:
  - **Undo**: Read the transaction's log backwards, issuing UNDO of each record

  - **Broadcast**: At each save point, invoke each RM telling it the transaction is at the savepoint

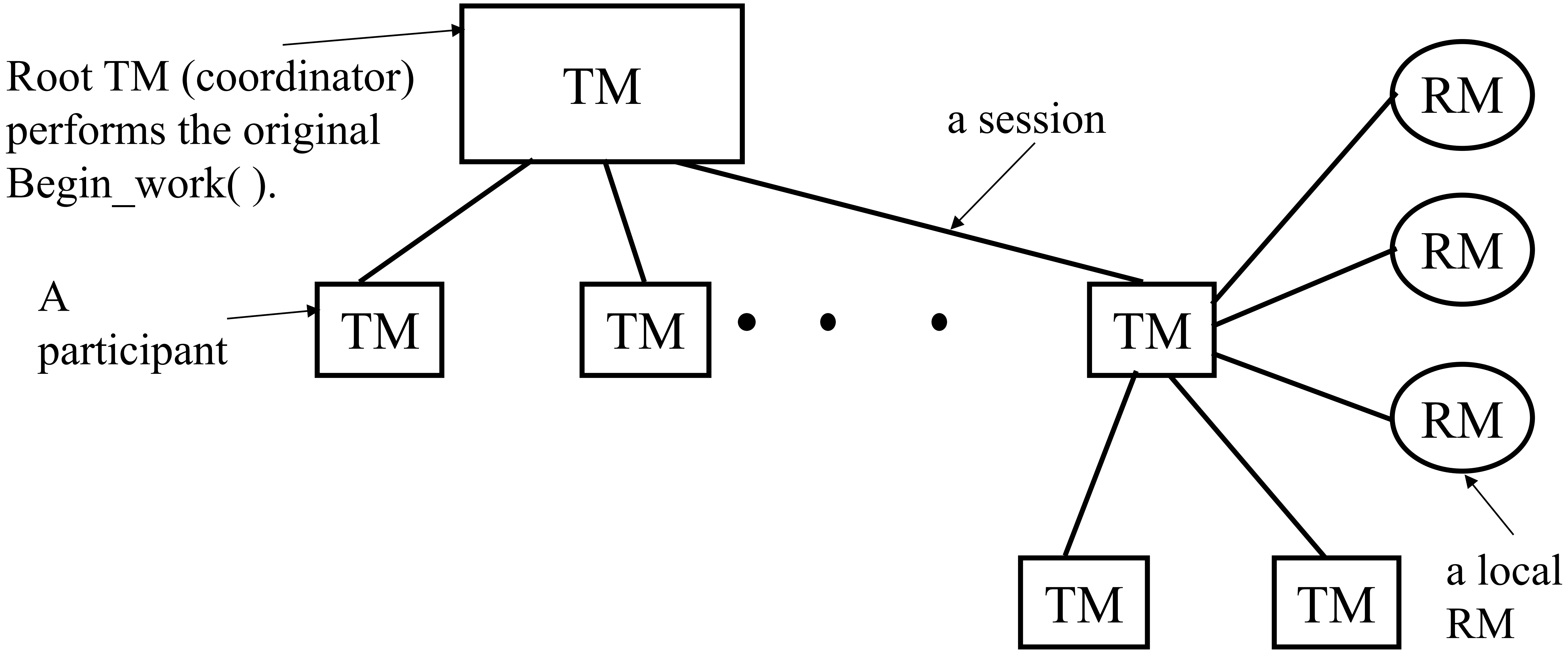
  - **Abort**: Write the transaction abort record to the log (UNDO of `Begin_Work()`)

  - **Complete**: Write a complete record to the log indicating that abort ended; Deallocate the live transaction state

Transaction Trees

- How does a transaction manager first hear about a **distributed** transaction?

- There are two cases:
  - Outgoing case: A local transaction sends a request to another node
  - Incoming case: A new transaction request arrives from a remote TM



Distributed Two-Phase Commit: Commit (1)

- **The root commit coordinator** executes the following logic when a successful `Commit_Work()` is invoked on a distributed transaction

- **Phase 1:**
  - **Local prepare**: Invoke each local RM to prepare for commit

  - **Distributed prepare**: Send a prepare request on each of the transaction's outgoing sessions

  - **Decide**: If all RM vote yes and all outgoing sessions respond yes, then durably write the transaction commit log record containing a list of a participating RMs and TMs

Distributed Two-Phase Commit: Commit (2)

- **Phase 2:**
  - **Commit**: Invoke each participating RM, telling it the commit decision; Send "commit" message on each of the transaction's outgoing sessions
  - **Complete**: When all local RMs and all outgoing sessions acknowledge commit, write a completion record to the log; When the completion record is durable, defalcate the live transaction state

Distributed Two-Phase Commit: Abort

- If a local RM or outgoing session votes **no** during phase 1, or if the prepare request waits longer than a **timeout** period, then the participant can abort the transaction and vote no to the coordinator

- The logic for abort:
  - **Broadcast**: Send abort message on all outgoing sessions
  - **Undo**: Scan the transaction log backwards; For each record, invoke the undo callback of the record's RM
  - **Complete**: Write a complete record to the log indicating that abort completed; Deallocate the live transaction state

Reference

[1] Jim Gray and Andreas Reuter, “Transaction Processing: Concepts and Techniques”, Morgan Kaufmann, San Mateo, CA (1993)

[2] Dr. Kien A. Hua, “Ch. 10. Transaction Manager Concepts - cs.ucf.edu”, http://www.cs.ucf.edu/courses/cop6730/notes/Ch10.ppt