# Transaction Management in the R* Distributed Database Management Systems

*- C. Mohan B. Lindsay and R. Obermarck, Dec 1986*

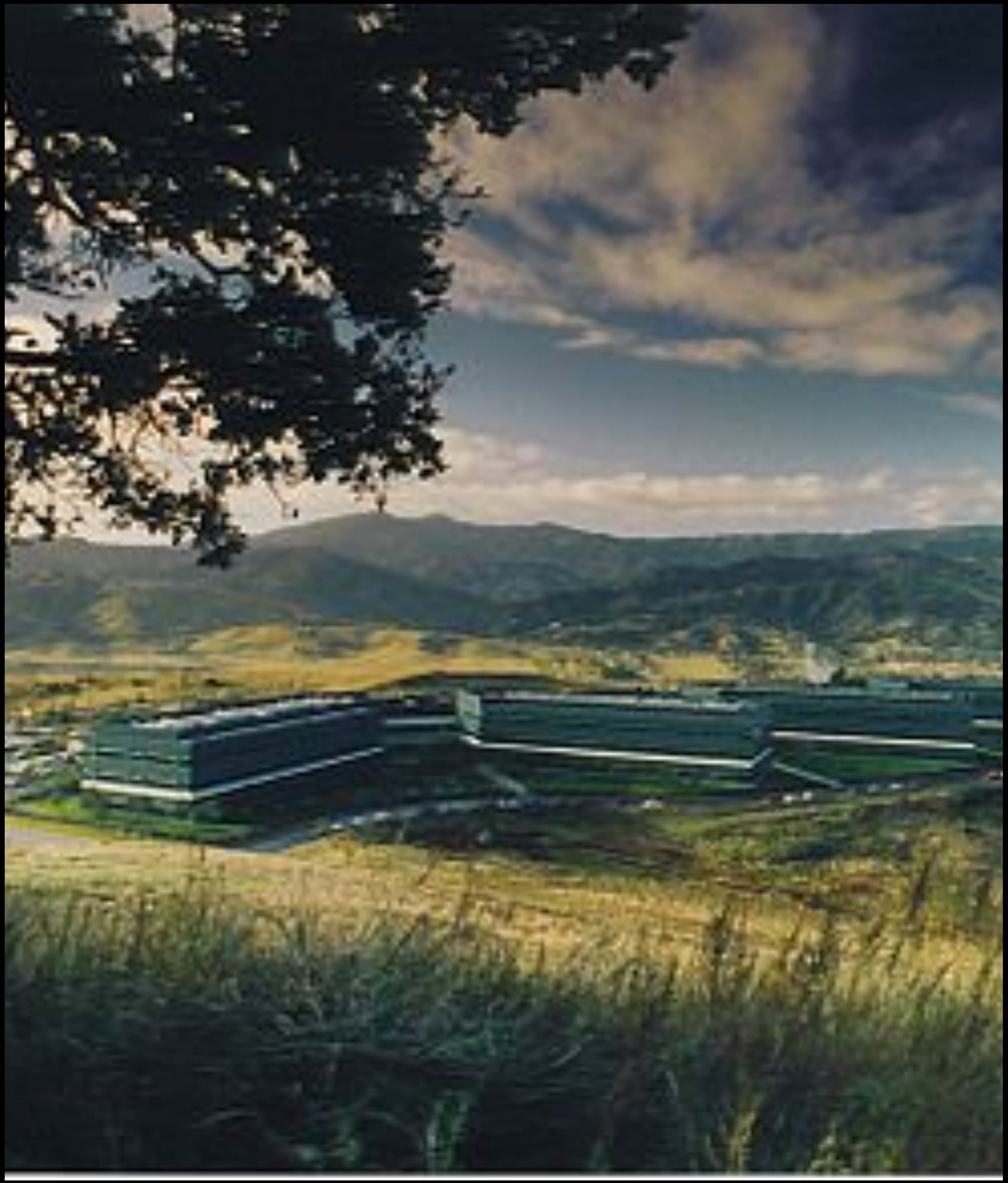
*Presented By*
*Shivani Teegala*
*Oct 4th '18 ECS 265A*

## OverView

- Introduction
  - Background
  - Assumptions & Terminology
  - Characteristics of CP
- Commit Protocol
  - 2P Commit Protocol
  - Hierarchical 2P
  - Presumed Abort
  - Presumed Commit
- Discussion
  - Performance Analysis
  - Blocking and Deadlock Management

## Background

- R* pronounced R star, is an experimental DDBMS developed out of IBM San Jose Research Laboratory
- R* is an evolution of System R and carry forwards the DBM , Concurrency control and 2PL from System R.
- *Fun Fact: The * denotes Kleene stars which means (ε,R,RR,RRR,RRR….)*



*“What if a transaction commits at one site and rolls back at another? Who guarantees the atomicity?”*

*“A distributed transaction commit protocol is required in order to ensure either all the effects of the transaction persist or that none of the effects persist…”*

# Transaction Manager

- Manages the commit protocol,
- Performs local and global deadlock detection,
- Assigns transaction Ids to new transactions.

## Characteristics of CP

- Always guarantee transaction atomicity
- Minimal overhead in terms of log writes and message traffic
- Optimised performance in no-failure case
- Exploitation of completely or partially read-only transaction
- Maximising the ability to perform unilateral aborts.

## Assumptions

- Transactions perform provisionally such that actions can be undone if needed.
- Each DB in DDBS has a log that is used to recoverably record the state of transaction.(UNDO/REDO log)
- Log records are written sequentially and kept in non - volatile storage.
- Transactions and processors are assumed to have globally unique names.

## Terminology

- Synchronous (Force-Write): Forced record and all preceding ones immediately moves from virtual memory buffers to Stable Storage.
  - Important to batch force-writes for high performance.
- Asynchronous (Write): Record gets written to virtual buffer storage and is allowed to migrate later.

## Two Phase Commit Protocol

*"In 2P, the model of a distributed transaction execution is such that there is one process, called the coordinator, that is connected to the user application and a set of other processes, called the subordinates. During the execution of the commit protocol the subordinates communicate only with the coordinator, not among themselves."*

The 'two phases' of 2PC are the *prepare* and the *commit* phase.

# Prepare - Coordinator

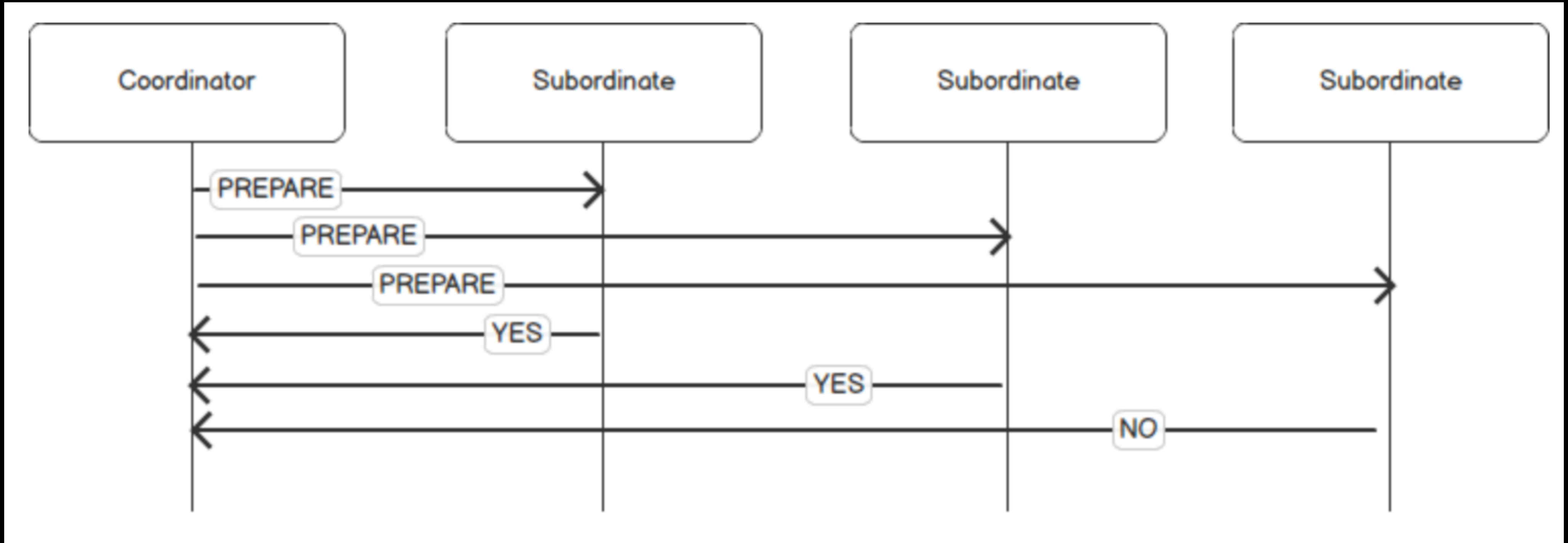


Figure 1. Coordinator prepare phase

- **Sends prepare Statements in Parallel**
- **Waits for the votes from Subordinate. Either one No or All Yes Votes.**

# Prepare - Subordinate

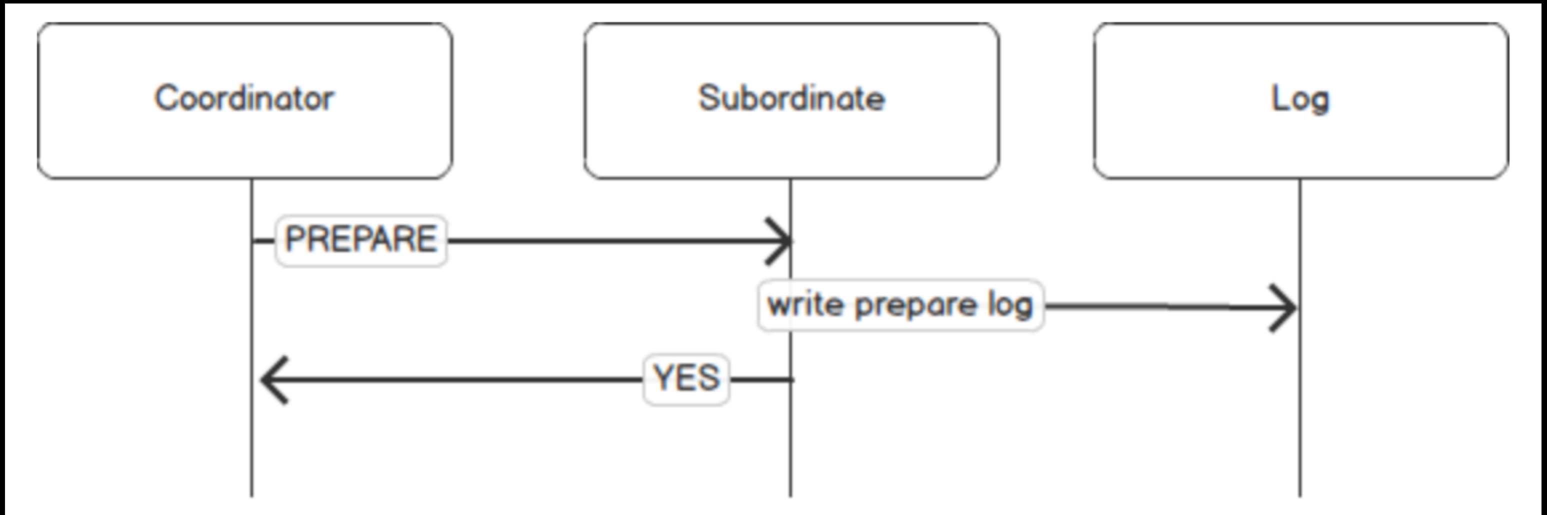


Figure 2. Subordinate prepare phase. YES vote.

- Writes force prepare log
- And sends the Yes Vote
- Enters Prepare State

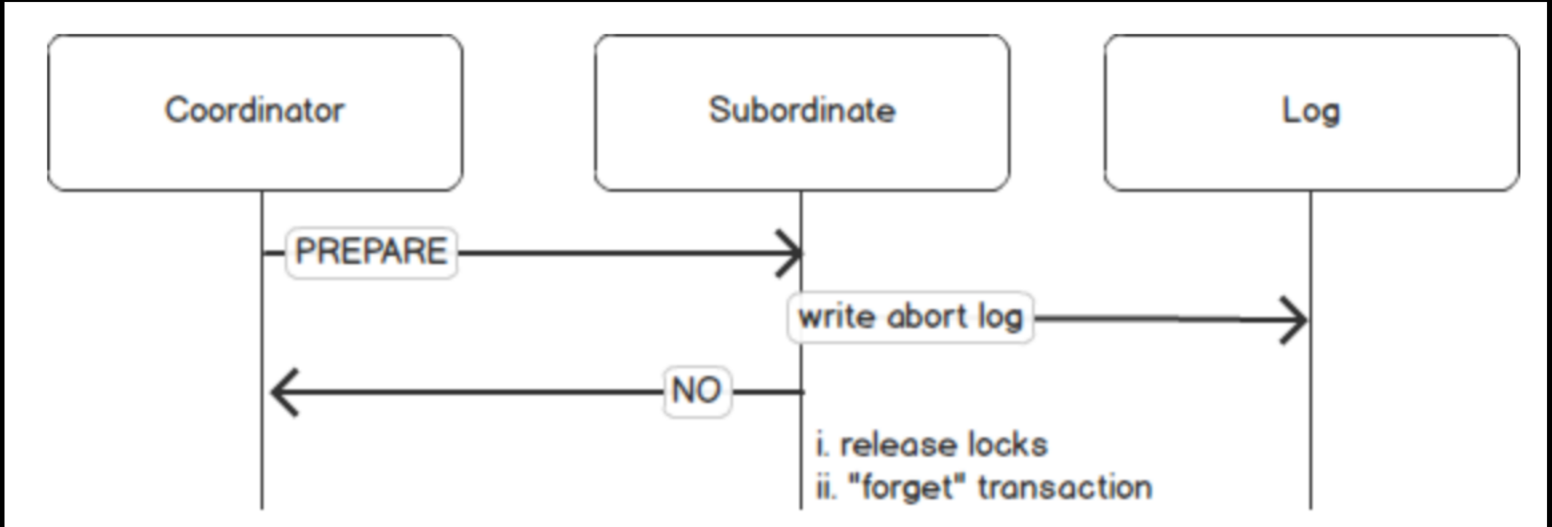


Figure 3. Subordinate prepare phase. NO vote.

- Writes forced abort log
- Sends Back No Vote
- Starts unilateral abort

# Commit Phase

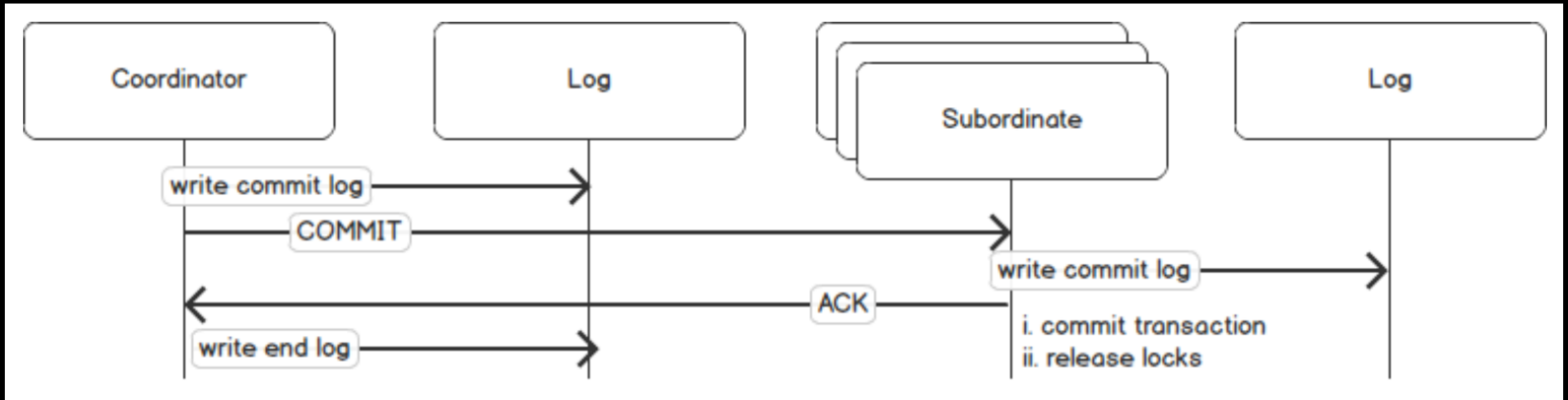


Figure 5. Commit phase. COMMIT transaction.

- Triggers after all votes are sent.

- Triggers immediately after at least one No Vote.
- Messages sent back only to Sub-ordinates who has not responded or responded as Yes.

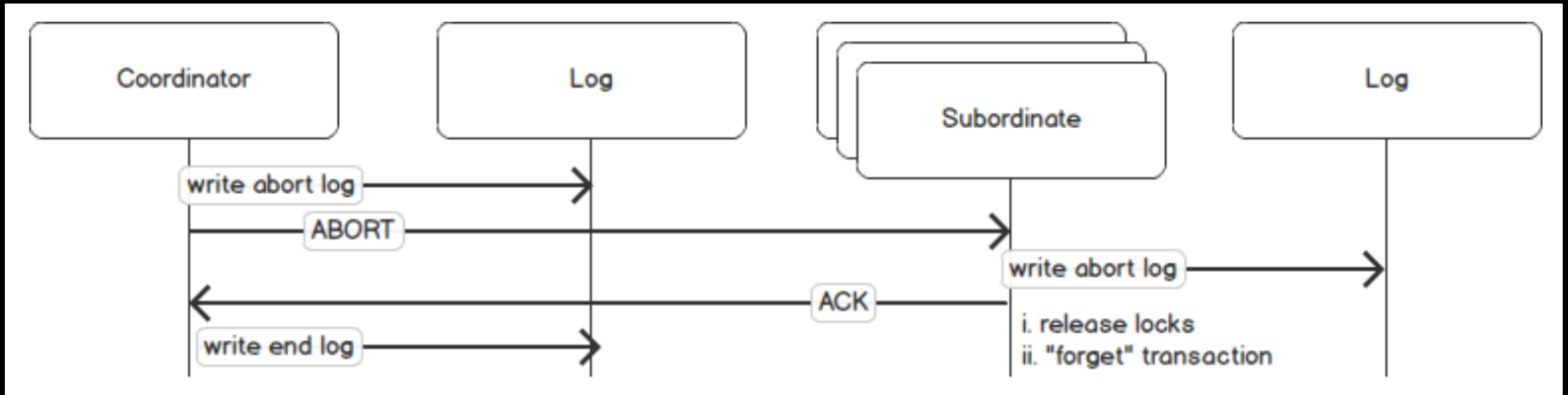
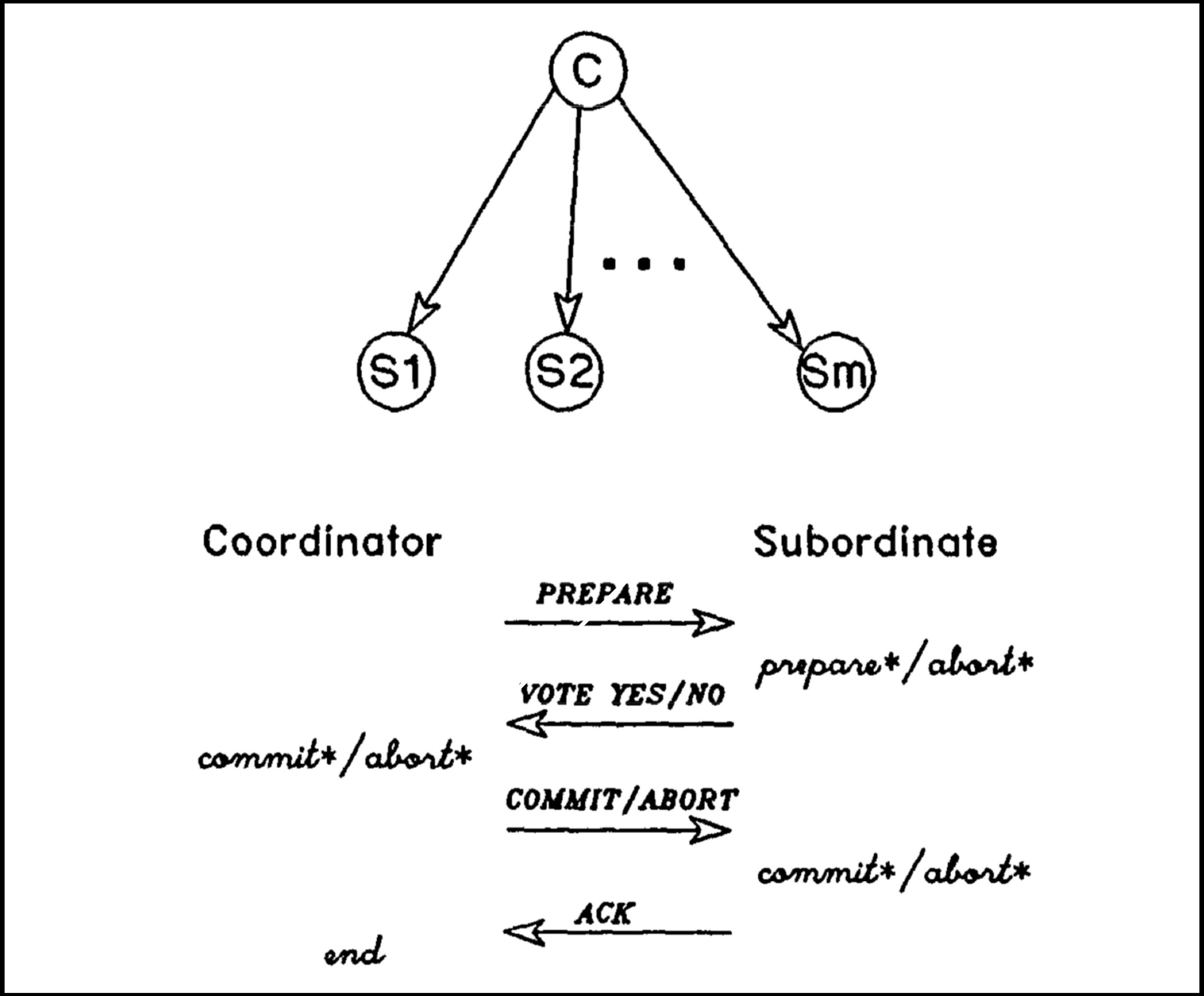


Figure 4. Commit phase. ABORT transaction.

- 2 Messages
- 2 Logs(1*)



- 2 Messages
- 2 logs(*)

** denotes force logs*

# Handling Failures

*“We assume that at each active site a recovery process exists and that it processes all messages from recovery processes at other sites and handles all the transactions that were executing the commit protocol at the time of the last failure of the site…”*

For each transaction executing at the time of the failure the recovery process determines whether:

- There are no 2PC protocol records of any kind, or
- The transaction is in either a committing or aborting state, or
- The transaction is in the prepared state (waiting for an outcome decision)

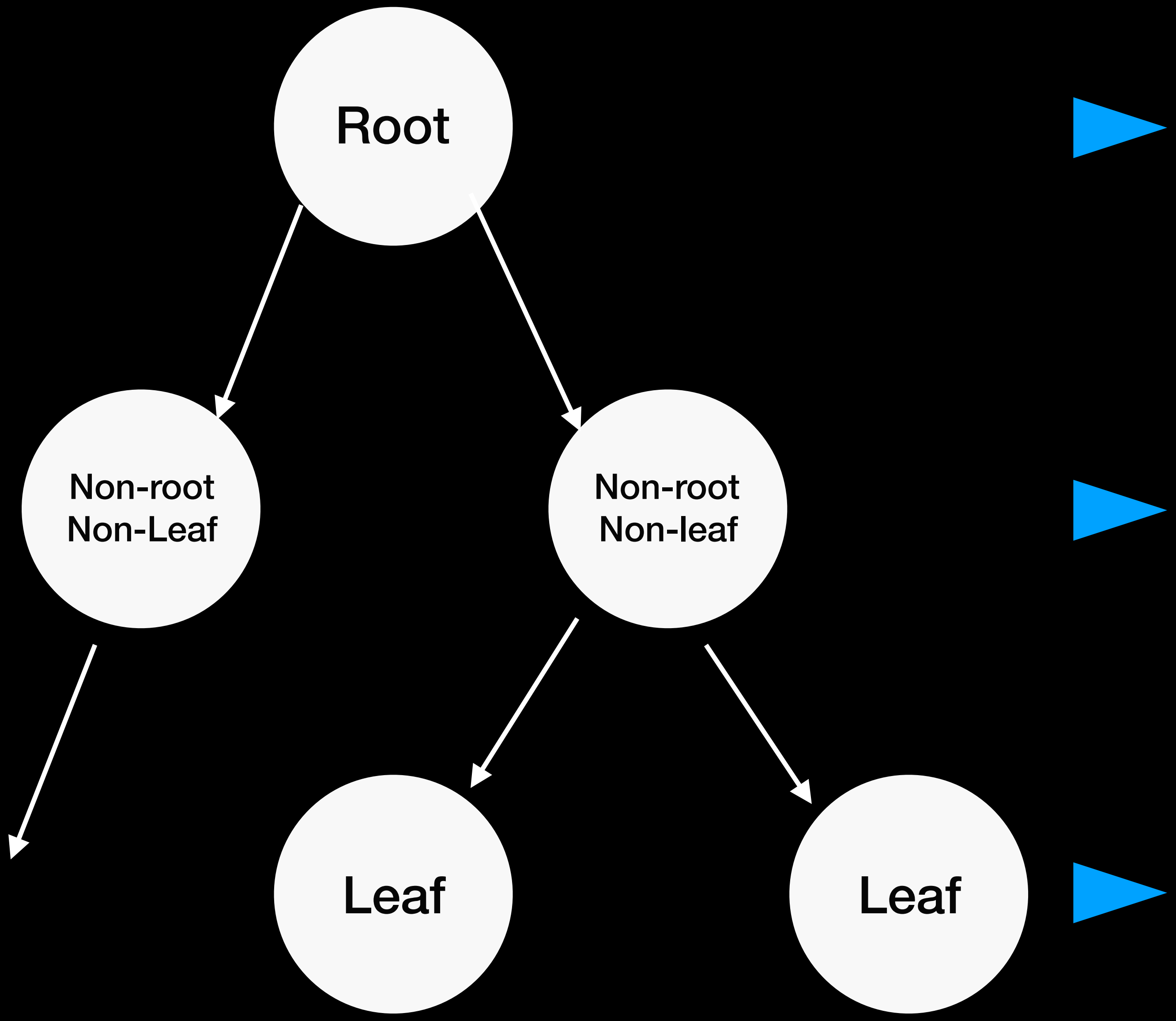
| Node | No Information | Prepared Log | Commit/Abort Log |
|---|---|---|---|
| **Coordiantor** | - Aborts the transaction | - | - Periodically sends commit/Abort msgs.<br>- Recovery Process takes over and performs normal protocol. |
| **Subordiante** | - Aborts the transaction | - Periodically tries to contact co-ordinator<br>- Recovery Process takes over and performs normal protocol. | - Reads the log.<br>- Recovery Process takes over and performs normal protocol. |

## “Why so many force-writes?”

**To ensure Transaction Atomicity**

*“By forcing their commit/abort records before sending the ACKs, the subordinates make sure that they will never be required (while recovering from a processor failure) to ask the coordinator about the final outcome after having acknowledged..”*

# Hierarchical 2P



- Only Co-ordinator
- Both Co-ordinator and Subordinate
- Sub ordinate

## Flow

- Root and leaf processes act as in regular 2PC.
- An intermediate node must propagate PREPAREs to its subordinates. It can vote YES only if all of its subordinates vote YES.
- In a similar manner, on receiving an ABORT or COMMIT an intermediate node must force-write its own commit (abort) record, send an ACK to the coordinator, and then propagate the decision to its subordinates.

## Presumed Abort & Presumed Commit

# Goals

- Always guarantee transaction atomicity
- **Minimal overhead in terms of log writes and message traffic**
- **Optimised performance in no-failure case**
- **Exploitation of completely or partially read-only transaction**
- **Maximising the ability to perform unilateral aborts.**

# Presumed Abort (PA)

**2PC — “In absence of any information — —> Abort”**

*“ The name arises from the fact that in the no information case the transaction is presumed to have aborted, and hence the recovery process’s response to an inquiry is an ABORT”*

This means that —> Safe to Immediately forget a transaction if decision is abort

- No Forced Abort records.
- No ACKs for aborts.
- No end record after an abort record.

# Read Only

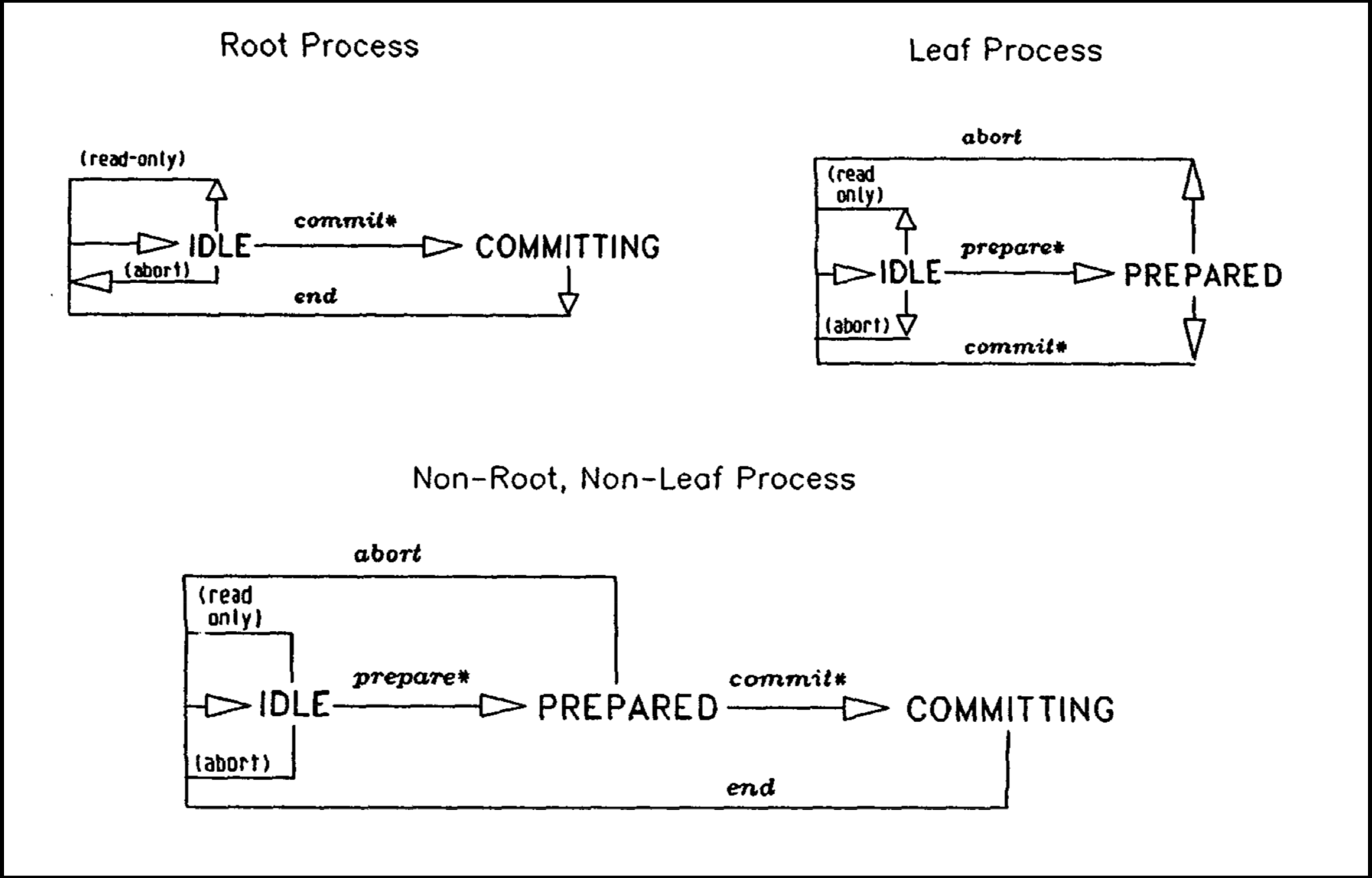
**For Read-Only transactions it doesn't matter, whether the transaction finally commits or not.**

| | | |
|---|---|---|
| Leaf Nodes | - Finds no UNDO/ Redo Logs<br>- Send READ VOTE | No Logs<br>1 Msg (Read Vote) |
| Non-root, Non-leaf Nodes | - Self and all subordinate's are Read Votes<br>- Sends READ VOTE | No Logs<br>1 Msg (Read Vote)<br>1 Msg (Prepare) |
| Root Node | - Coordinator is read-only and receives READ VOTES<br>- Transaction is READ ONLY | No Logs<br>1 Msg (Prepare) |

# Partial - Read Only

| | |
|---|---|
| Leaf Nodes | Send YES/NO VOTE<br>Logs Commit*<br>Sends ACK |
| Non-root, Non-leaf Nodes | Sends Prepare<br>Logs Prepare*<br>Sends YES/NO vote<br>Logs Commit*<br>Sends Commit to Non-Read Only<br>Sends ACK<br>Logs ends |
| Root Node | Sends Prepare<br>Logs Commit*<br>Sends Commit to Non-Read Only<br>Logs End |

## State Changes and Log writes - PA



*Information in parentheses indicates under what circumstances such transitions take place. IDLE is the initial and final state for each process*

# Presumed Commit

Generally, Are the transactions expected to be Committed or Aborted?    Commited

Makes more sense to

- ACK Aborts
- Force Abort logs by subordinates.
- Incase of No Information ——> Assume Commit

But there is a small problem with this…

What if Root Process crashes before sending commit or abort message?

# Contd.

**Collecting State**:

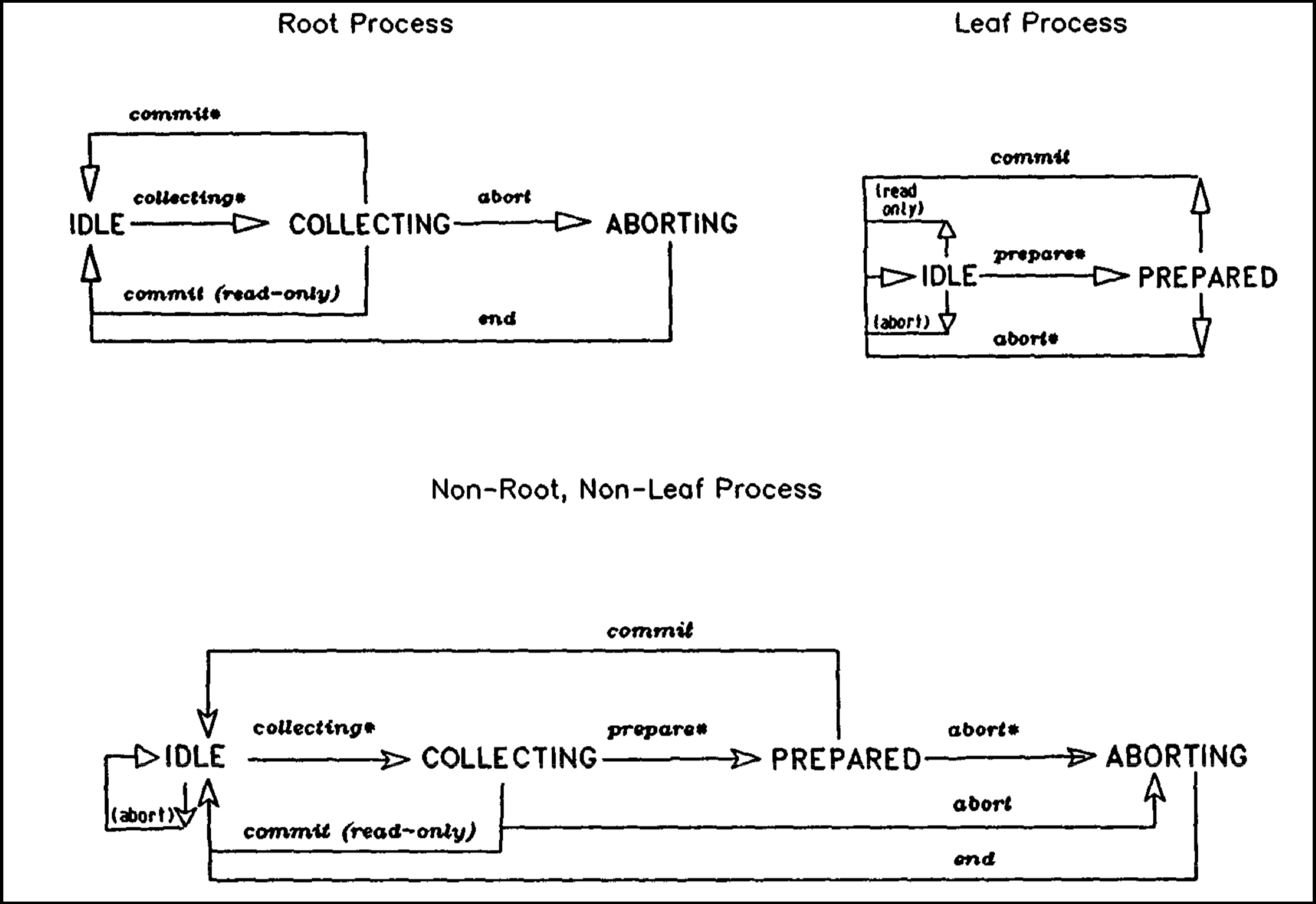
Co-ordinator records information on sub-ordinates safely before sending the prepares.

— Incase the recovery process finds, collecting record and no other following it, it force aborts and informs all subordinates and gets ACKS.

| **PC** | **PA** |
|---|---|
| Assumed Commit | Assumed Abort |
| Collecting State in First Phase | No Collecting state |
| Force writes Aborts (Except root process) | Force Writes Commits |
| ACK for Aborts | ACK for Commits |
| Writes Commit log for read-only | No logs for read only |

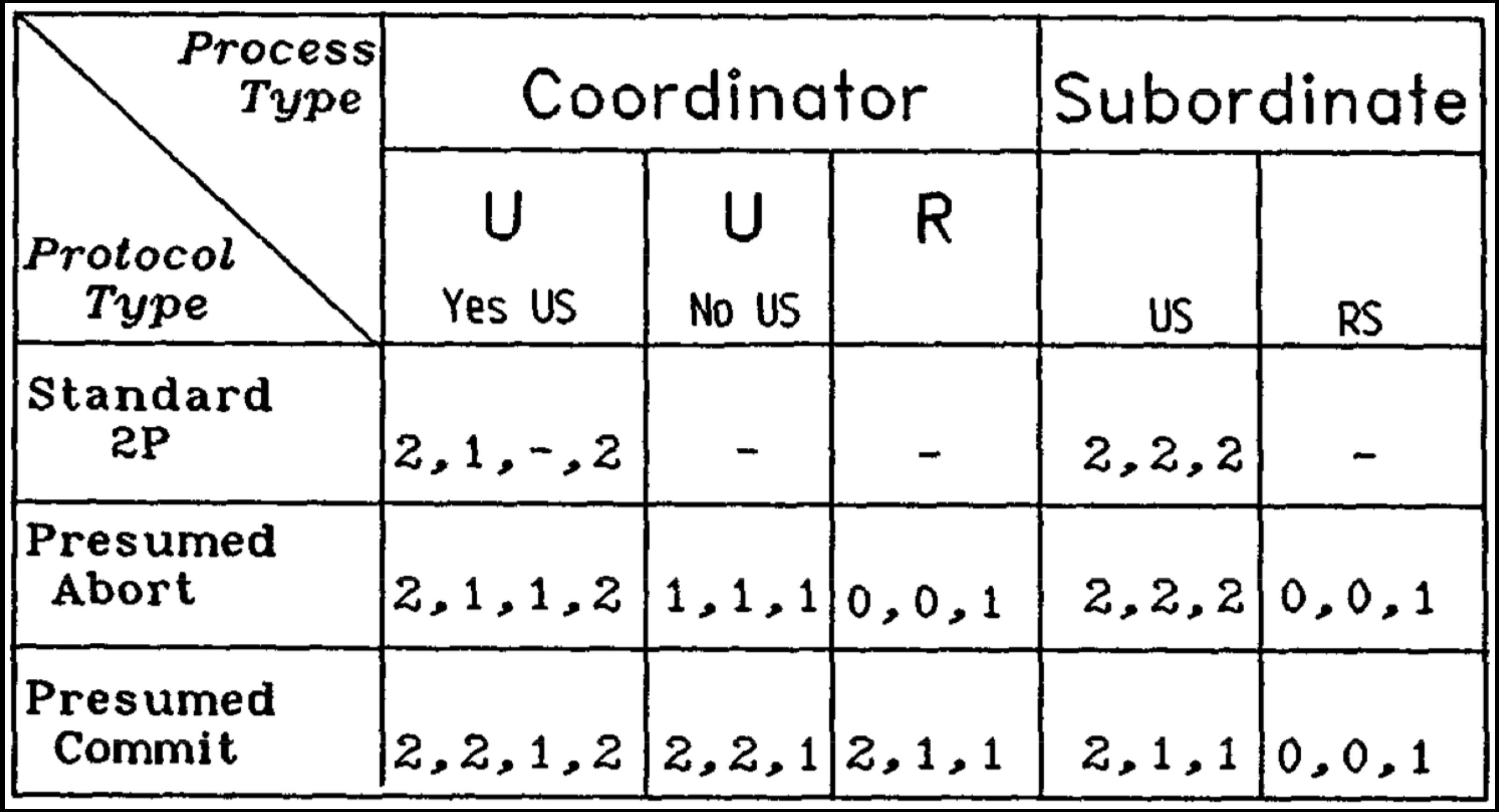
# PC (Cont.)

| | Read Only | Partial Read-Only |
|---|---|---|
| **Leaf** | Sends READ VOTE | Prepare Log*<br>Sends Yes Vote<br>Commit Log |
| **Non-Leaf Non Root** | Collecting Log*<br>Sends Prepare<br>Commit Log<br>Sends READ VOTE | Collecting Log*<br>Prepare Log*<br>Sends Prepare<br>Sends Yes Vote<br>Commit Log<br>Sends Commit for Non-Read |
| **Root** | Collecting Log*<br>Sends Prepare<br>Commit Log | Collecting Log*<br>Sends Prepare<br>Commit Log*<br>Sends Commit for Non-Read |

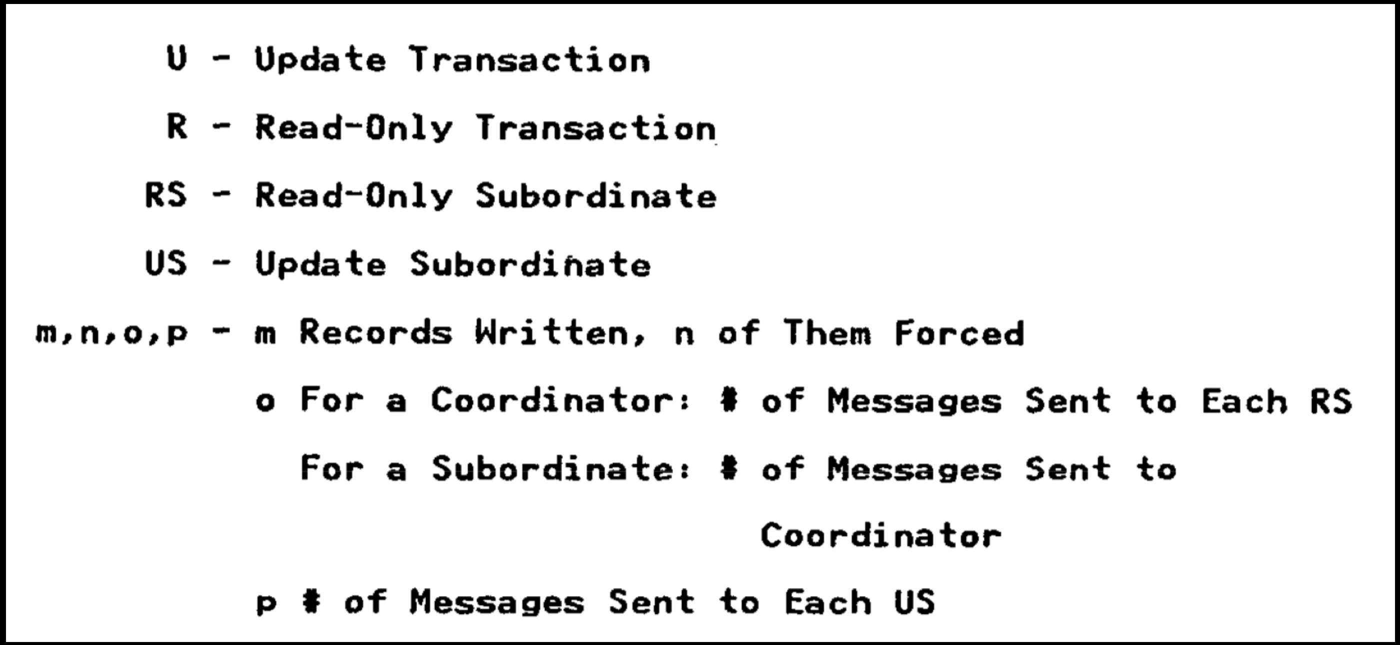


**Information in parentheses indicates under what circumstances such transitions take place. IDLE is the initial and final state for each process.**

# Performance Evaluation



| *Process Type* / *Protocol Type* | Coordinator | | | Subordinate | |
|---|---|---|---|---|---|
| | U<br>Yes US | U<br>No US | R | US | RS |
| Standard 2P | 2,1,-,2 | - | - | 2,2,2 | - |
| Presumed Abort | 2,1,1,2 | 1,1,1 | 0,0,1 | 2,2,2 | 0,0,1 |
| Presumed Commit | 2,2,1,2 | 2,2,1 | 2,1,1 | 2,1,1 | 0,0,1 |



| | |
|---|---|
| U - | Update Transaction |
| R - | Read-Only Transaction |
| RS - | Read-Only Subordinate |
| US - | Update Subordinate |
| m,n,o,p - | m Records Written, n of Them Forced |
| | o For a Coordinator: # of Messages Sent to Each RS |
| | For a Subordinate: # of Messages Sent to Coordinator |
| | p # of Messages Sent to Each US |

## Discussion

| | 2P | PA | PC |
|---|---|---|---|
| Read Only | - | Better | - |
| Partial Read Only(Only co-ordinator Updates) | - | Better | - |
| Partial Read Only( With Update Sub ordinates) | - | - | Better |

# Blocking and DeadLocks

*“We have extended, but not implemented, PA and PC to reduce the probability of blocking by allowing a prepared process.. ”*

Process might wait for one of two reasons:

- To obtain a lock and
- To receive a message from a cohort process of the same transaction

**Each DD wakes up periodically and looks for deadlocks after gathering the wait-for information from the local DBMS and the communication manager**

**- To break the cycle generally a local victim is chosen.**

# References

https://blog.acolyer.org/2016/01/11/transaction-management-in-r/

https://blog.acolyer.org/2016/01/12/presume-abort-commit/

https://people.eecs.berkeley.edu/~fox/summaries/database/rstar_trans.html

https://pdfs.semanticscholar.org/06e2/5c1f69155e53af51170c08687e1dcf272974.pd

https://sookocheff.com/post/databases/distributed-transaction-management/