Chapter 10: Distributed DBMS Reliability

- Definitions and Basic Concepts
- Local Recovery Management
- In-place update, out-of-place update
- Distributed Reliability Protocols
- Two phase commit protocol
- Three phase commit protocol

Acknowledgements: I am indebted to Arturas Mazeika for providing me his slides of this course.

Reliability

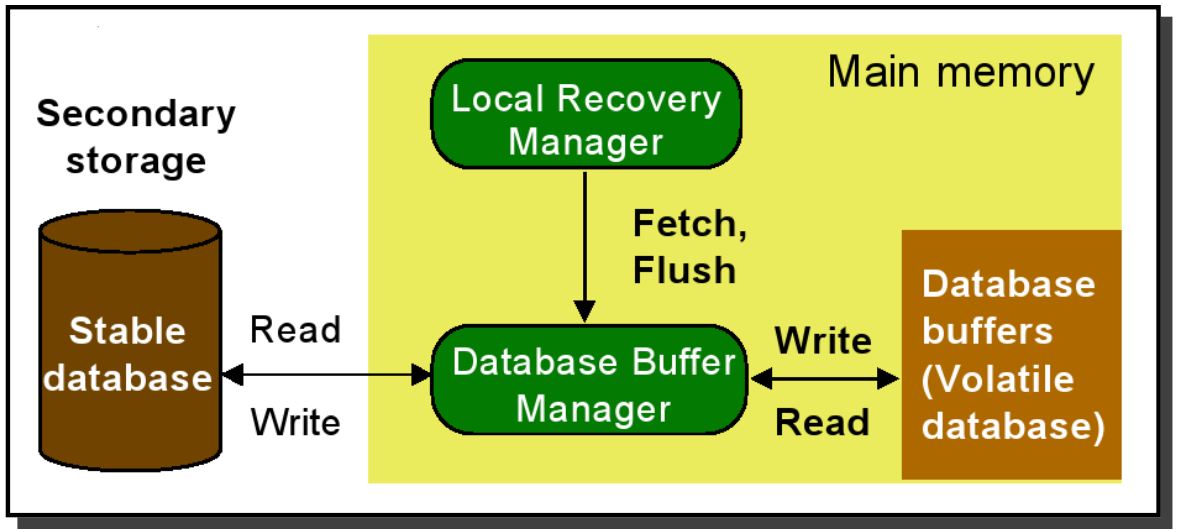
- A **reliable DDBMS** is one that can continue to process user requests even when the underlying system is unreliable, i.e., failures occur
- **Failures**
  - Transaction failures
  - System (site) failures, e.g., system crash, power supply failure
  - Media failures, e.g., hard disk failures
  - Communication failures, e.g., lost/undeliverable messages
- Reliability is closely related to the problem of how to maintain the **atomicity** and **durability** properties of transactions

Reliability . . .

- **Recovery system:** Ensures atomicity and durability of transactions in the presence of failures (and concurrent transactions)
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the DB contents to a state that ensures atomicity, consistency and durability

Local Recovery Management

- The **local recovery manager (LRM)** maintains the atomicity and durability properties of local transactions at each site.
- **Architecture**
  - **Volatile** storage: The main memory of the computer system (RAM)
  - **Stable** storage
    - A storage that “never” looses its contents
    - In reality this can only be approximated by a combination of hardware (non-volatile storage) and software (stable-write, stable-read, clean-up) components

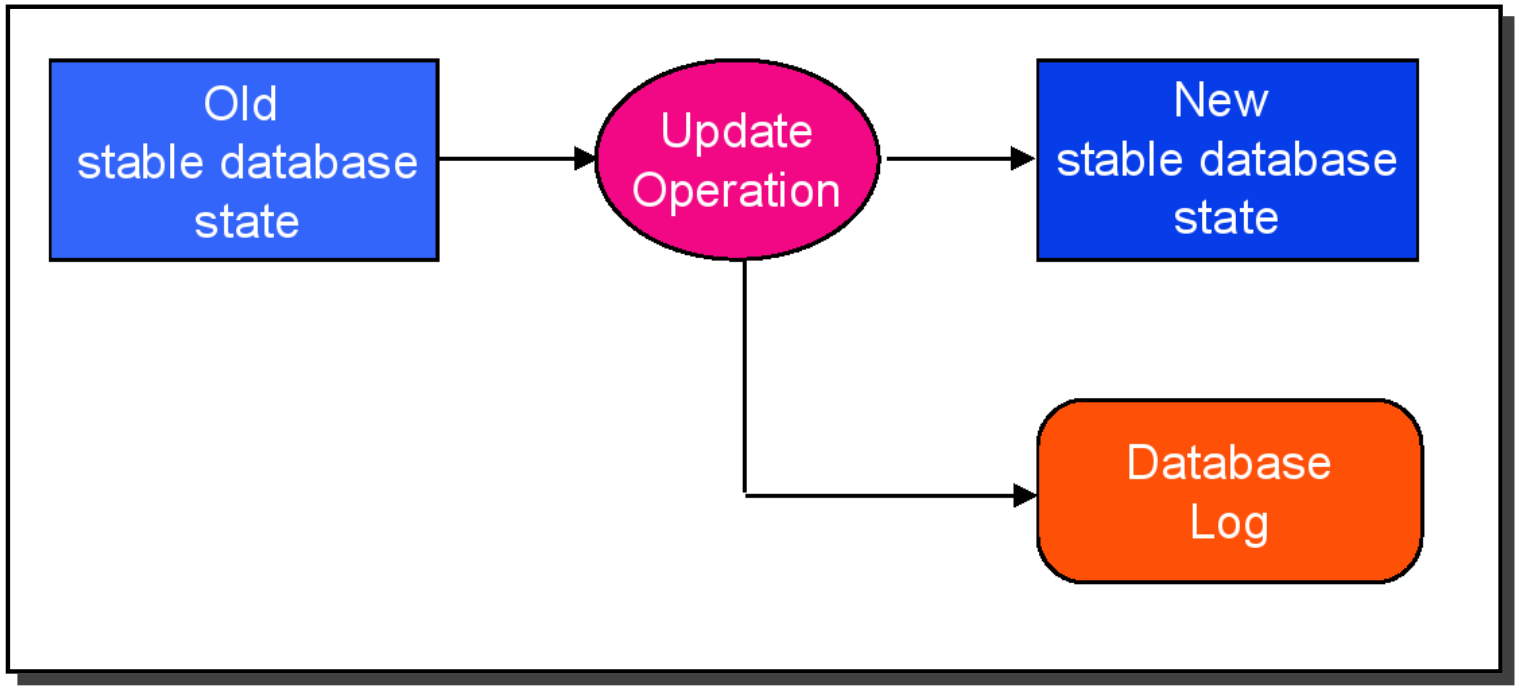


Local Recovery Management . . .

- Two ways for the LRM to deal with update/write operations
  - **In-place update**
    - Physically changes the value of the data item in the stable database
    - As a result, previous values are lost
    - Mostly used in databases
  - **Out-of-place update**
    - The new value(s) of updated data item(s) are stored separately from the old value(s)
    - Periodically, the updated values have to be integrated into the stable DB

In-Place Update

- Since in-place updates cause previous values of the affected data items to be lost, it is necessary to keep enough information about the DB updates in order to allow recovery in the case of failures
- Thus, every action of a transaction must not only perform the action, but must also write a log record to an append-only **log file**

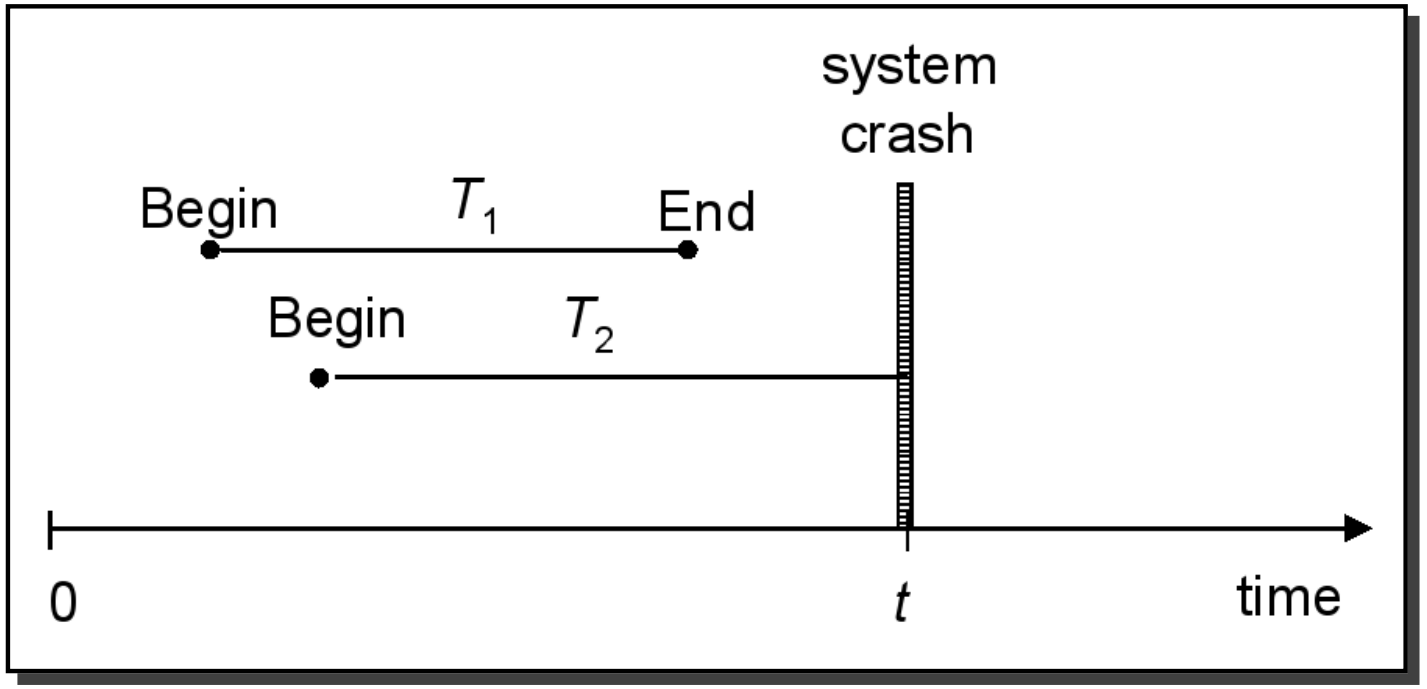


In-Place Update ...

- A **log** is the most popular structure for recording DB modifications on stable storage
- Consists of a sequence of **log records** that record all the update activities in the DB
- Each log record describes a significant event during transaction processing
- Types of log records
  - $< T_i, \mathbf{start} >$: if transaction T_i has started
  - $< T_i, X_j, V_1, V_2 >$: before T_i executes a $\mathbf{write}(X_j)$, where V_1 is the old value before the write and V_2 is the new value after the write
  - $< T_i, \mathbf{commit} >$: if T_i has committed
  - $< T_i, \mathbf{abort} >$: if T_i has aborted
  - $< \mathbf{checkpoint} >$
- With the information in the log file the recovery manager can restore the consistency of the DB in case of a failure.

In-Place Update ...

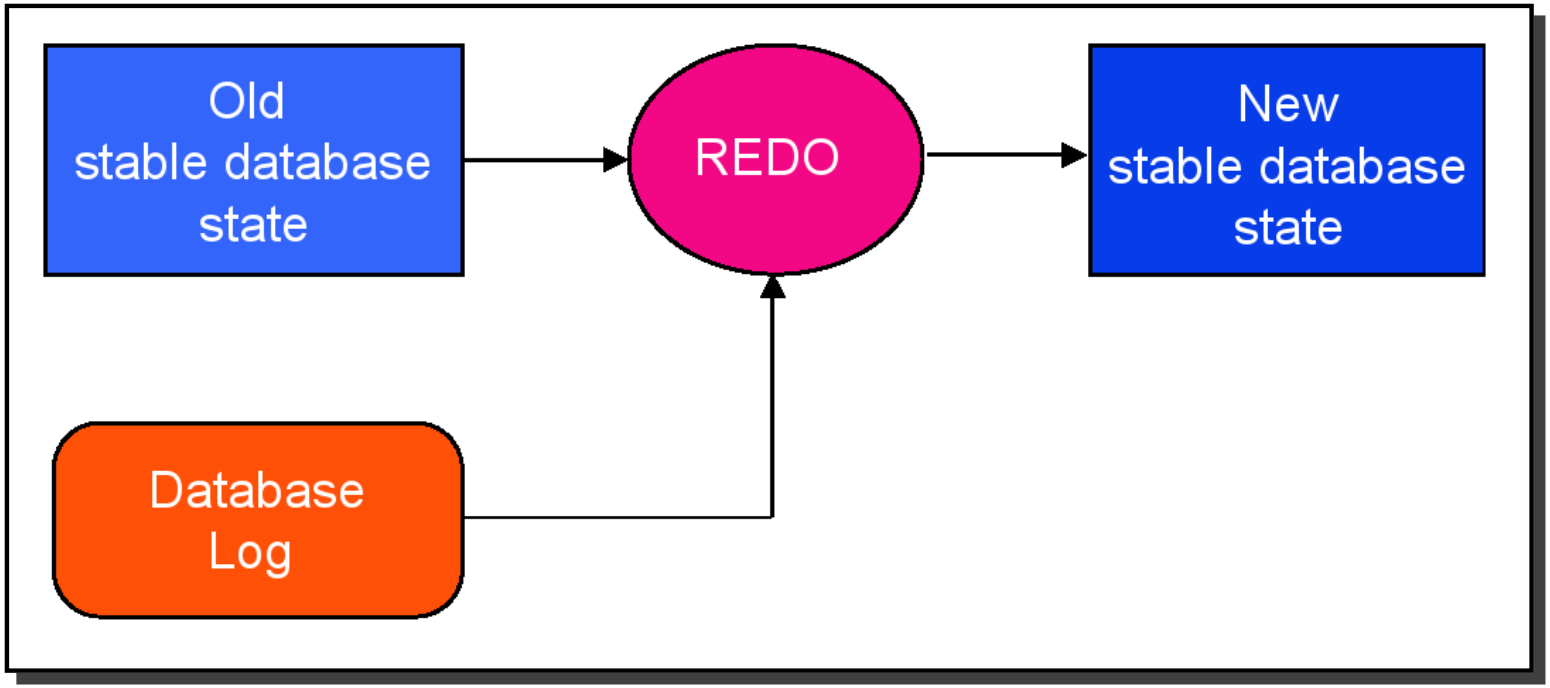
- Assume the following situation when a system crash occurs



- Upon recovery:
  - All effects of transaction T_1 should be reflected in the database ($\Rightarrow$ REDO)
  - None of the effects of transaction T_2 should be reflected in the database ($\Rightarrow$ UNDO)

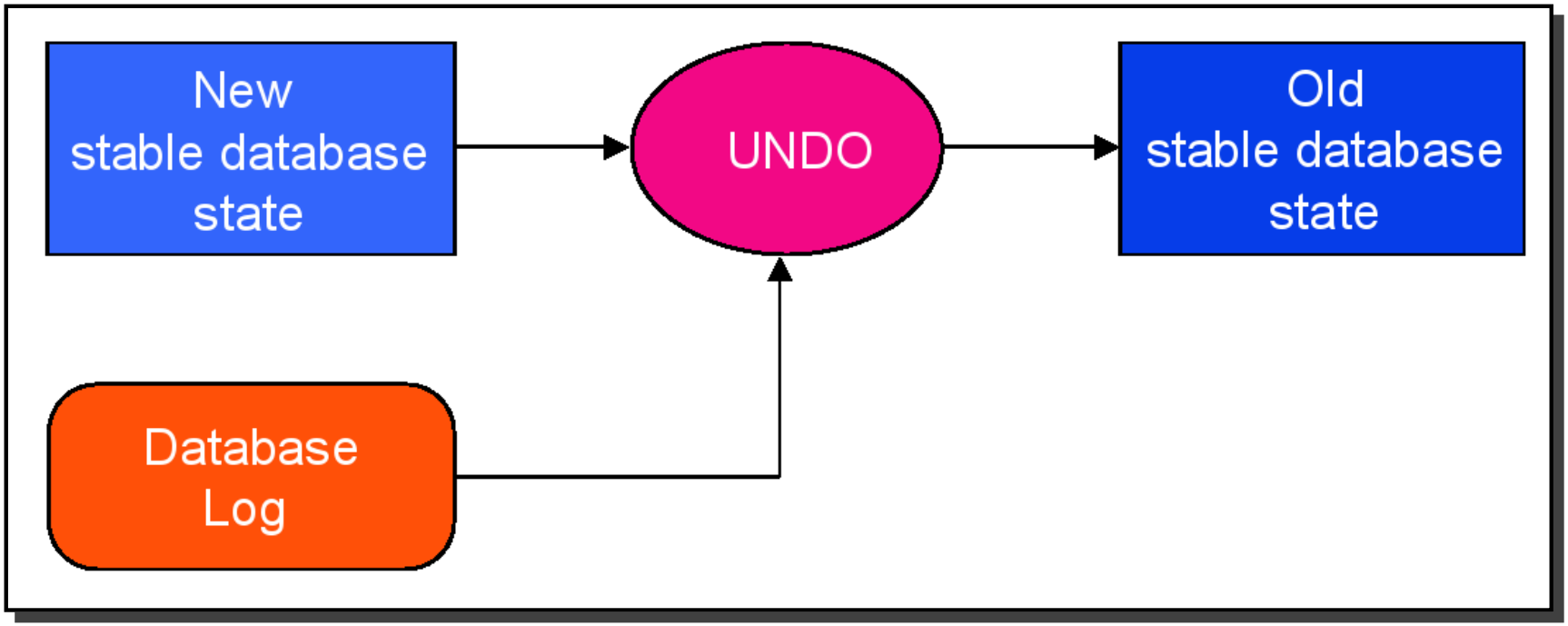
In-Place Update . . .

- **REDO Protocol**
  - REDO'ing an action means performing it again
  - The REDO operation uses the log information and performs the action that might have been done before, or not done due to failures
  - The REDO operation generates the new image



In-Place Update . . .

- **UNDO Protocol**
  - UNDO'ing an action means to restore the object to its image before the transaction has started
  - The UNDO operation uses the log information and restores the old value of the object



In-Place Update ...

- **Example:** Consider the transactions T_0 and T_1 (T_0 executes before T_1) and the following initial values: $A = 1000$, $B = 2000$, and $C = 700$

T_0: $read(A)$
$A = A - 50$
$write(A)$
$read(B)$
$B = B + 50$
$write(B)$

T_1: $read(C)$
$C = C - 100$
$write(C)$

  - Possible order of actual outputs to the log file and the DB:

Log	DB
$< T_0, start >$	
$< T_0, A, 1000, 950 >$	
$< T_0, B, 2000, 2050 >$	
$< T_0, commit >$	
	$A = 950$
	$B = 2050$
$< T_1, start >$	
$< T_1, C, 700, 600 >$	
$< T_1, commit >$	
	$C = 600$

In-Place Update ...

- **Example (contd.):** Consider the log after some system crashes and the corresponding recovery actions

(a) $< T_0, start >$
$< T_0, A, 1000, 950 >$
$< T_0, B, 2000, 2050 >$

(b) $< T_0, start >$
$< T_0, A, 1000, 950 >$
$< T_0, B, 2000, 2050 >$
$< T_0, commit >$
$< T_1, start >$
$< T_1, C, 700, 600 >$

(c) $< T_0, start >$
$< T_0, A, 1000, 950 >$
$< T_0, B, 2000, 2050 >$
$< T_0, commit >$
$< T_1, start >$
$< T_1, C, 700, 600 >$
$< T_1, commit >$

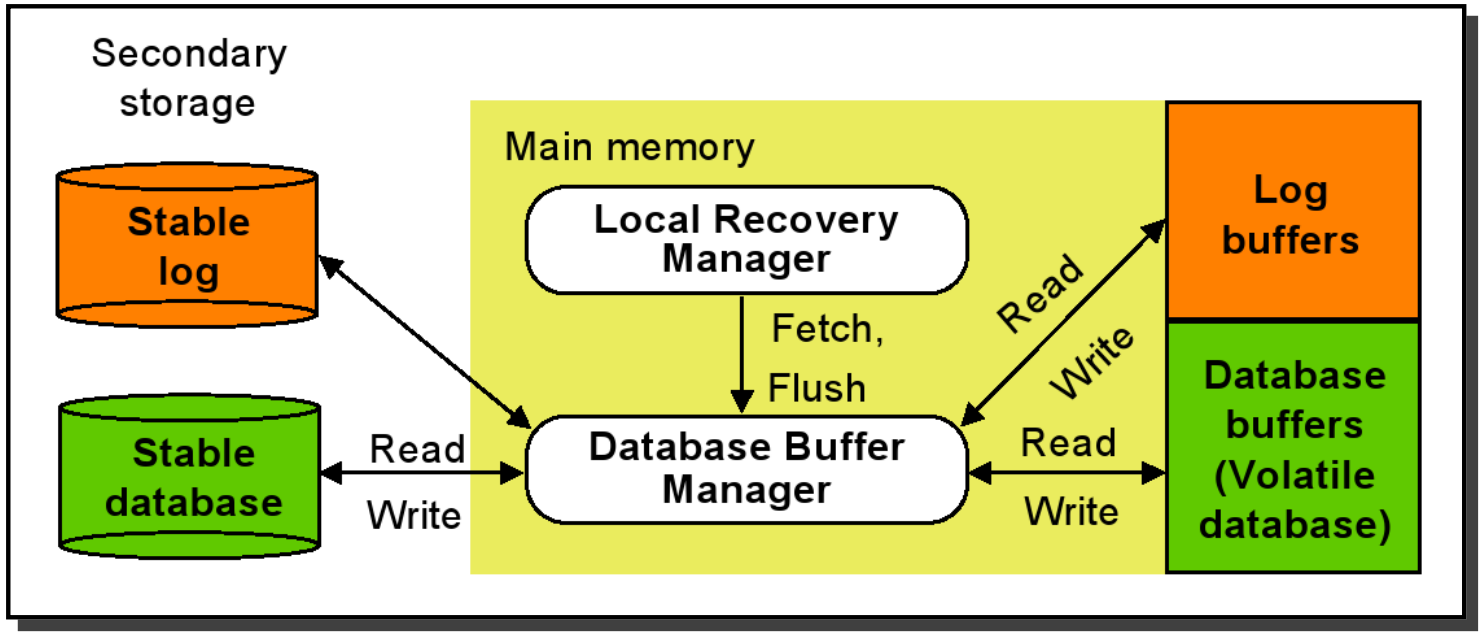
(a) undo(T0): B is restored to 2000 and A to 1000

(b) undo(T1) and redo(T0): C is restored to 700, and then A and B are set to 950 and 2050, respectively

(c) redo(T0) and redo(T1): A and B are set to 950 and 2050, respectively; then C is set to 600

In-Place Update . . .

- **Logging Interface**



- Log pages/buffers can be written to stable storage in two ways:
  - **synchronously**
    - The addition of each log record requires that the log is written to stable storage
    - When the log is written synchronoously, the execuction of the transaction is supended until the write is complete $\longrightarrow$ delay in response time
  - **asynchronously**
    - Log is moved to stable storage either at periodic intervals or when the buffer fills up.

In-Place Update . . .

- **When to write** log records into stable storage?
- Assume a transaction T updates a page P
- Fortunate case
  - System writes P in stable database
  - System updates stable log for this update
  - SYSTEM FAILURE OCCURS!... (before T commits)
  - We can recover (undo) by restoring P to its old state by using the log
- Unfortunate case
  - System writes P in stable database
  - SYSTEM FAILURE OCCURS!... (before stable log is updated)
  - We cannot recover from this failure because there is no log record to restore the old value
- Solution: Write-Ahead Log (WAL) protocol

In-Place Update . . .

- Notice:
  - If a system crashes before a transaction is committed, then all the operations must be undone. We need only the before images (undo portion of the log)
  - Once a transaction is committed, some of its actions might have to be redone. We need the after images (redo portion of the log)
- **Write-Ahead-Log (WAL) Protocol**
  - Before a stable database is updated, the undo portion of the log should be written to the stable log
  - When a transaction commits, the redo portion of the log must be written to stable log prior to the updating of the stable database

Out-of-Place Update

- Two out-of-place strategies are shadowing and differential files
- **Shadowing**
  - When an update occurs, don't change the old page, but create a shadow page with the new values and write it into the stable database
  - Update the access paths so that subsequent accesses are to the new shadow page
  - The old page is retained for recovery
- **Differential files**
  - For each DB file F maintain
    - a read-only part FR
    - a differential file consisting of insertions part (DF^+) and deletions part (DF^-)
  - Thus, $F = (FR \cup DF^+) - DF^-$

Distributed Reliability Protocols

- As with local reliability protocols, the distributed versions aim to maintain the atomicity and durability of distributed transactions
- Most problematic issues in a distributed transaction are commit, termination, and recovery
  - **Commit protocols**
    - How to execute a commit command for distributed transactions
    - How to ensure atomicity (and durability)?
  - **Termination protocols**
    - If a failure occurs at a site, how can the other operational sites deal with it
    - **Non-blocking:** the occurrence of failures should not force the sites to wait until the failure is repaired to terminate the transaction
  - **Recovery protocols**
    - When a failure occurs, how do the sites where the failure occurred deal with it
    - **Independent:** a failed site can determine the outcome of a transaction without having to obtain remote information

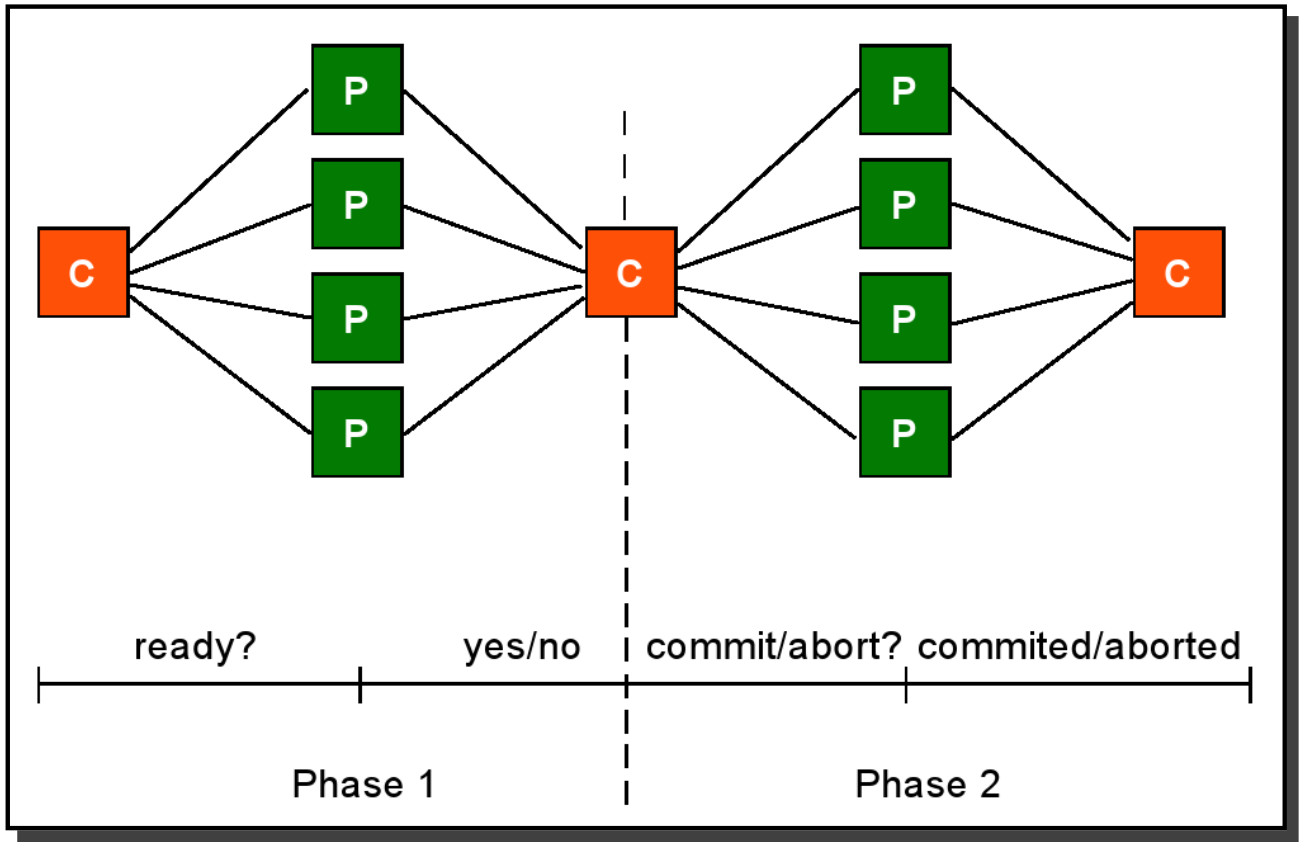
Commit Protocols

- Primary requirement of commit protocols is that they maintain the atomicity of distributed transactions (**atomic commitment**)
  - i.e., even though the exectution of the distributed transaction involves multiple sites, some of which might fail while executing, the effects of the transaction on the distributed DB is all-or-nothing.
- In the following we distinguish two roles
  - Coordinator: The process at the site where the transaction originates and which controls the execution
  - Participant: The process at the other sites that participate in executing the transaction

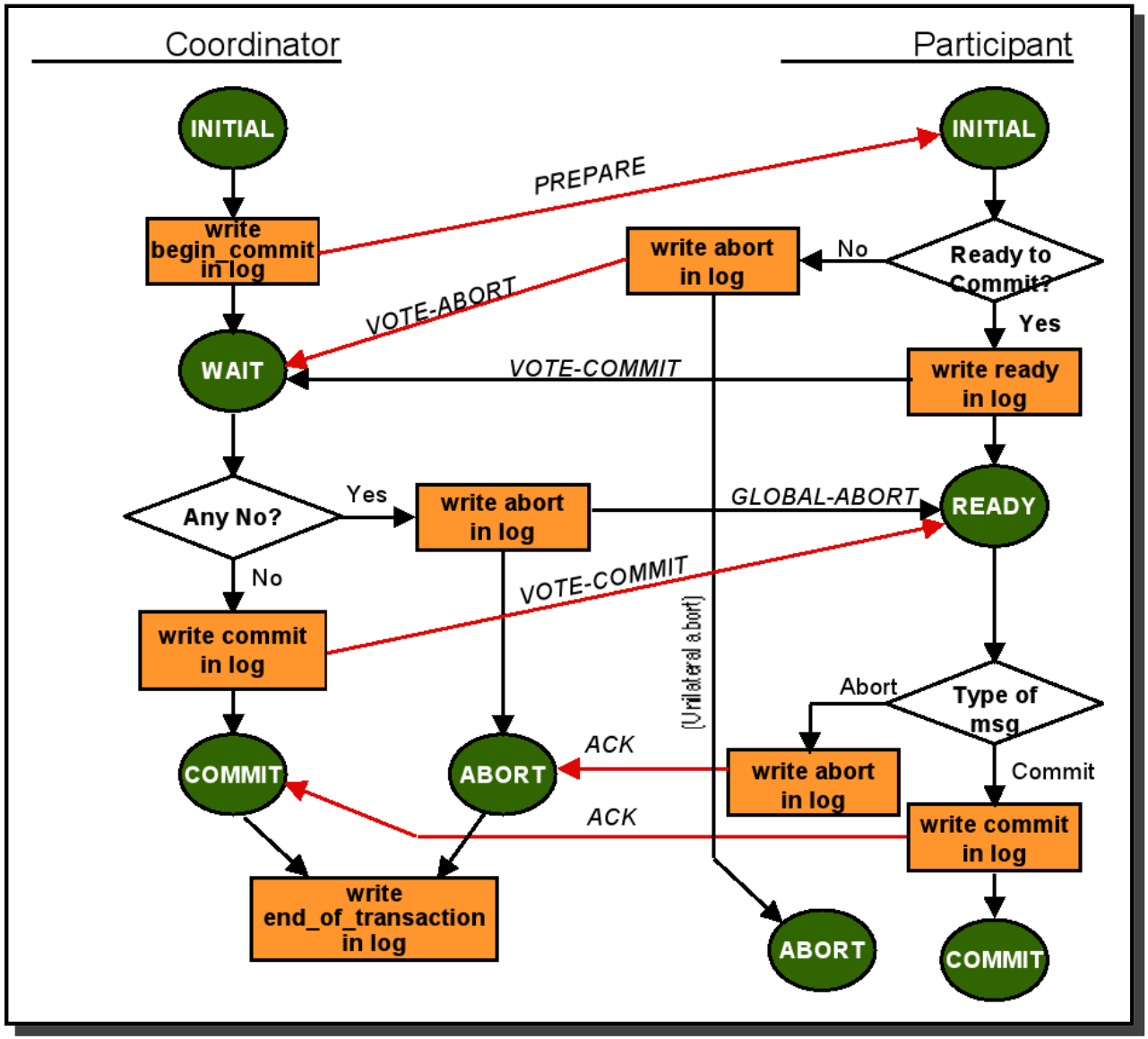
Centralized Two Phase Commit Protocol (2PC)

- Very simple protocol that ensures the atomic commitment of distributed transactions.
- **Phase 1:** The coordinator gets the participants ready to write the results into the database
- **Phase 2:** Everybody writes the results into the database
- **Global Commit Rule**
  - The coordinator aborts a transaction if and only if at least one participant votes to abort it
  - The coordinator commits a transaction if and only if all of the participants vote to commit it
- **Centralized** since communication is only between coordinator and the participants

Centralized Two Phase Commit Protocol (2PC) . . .

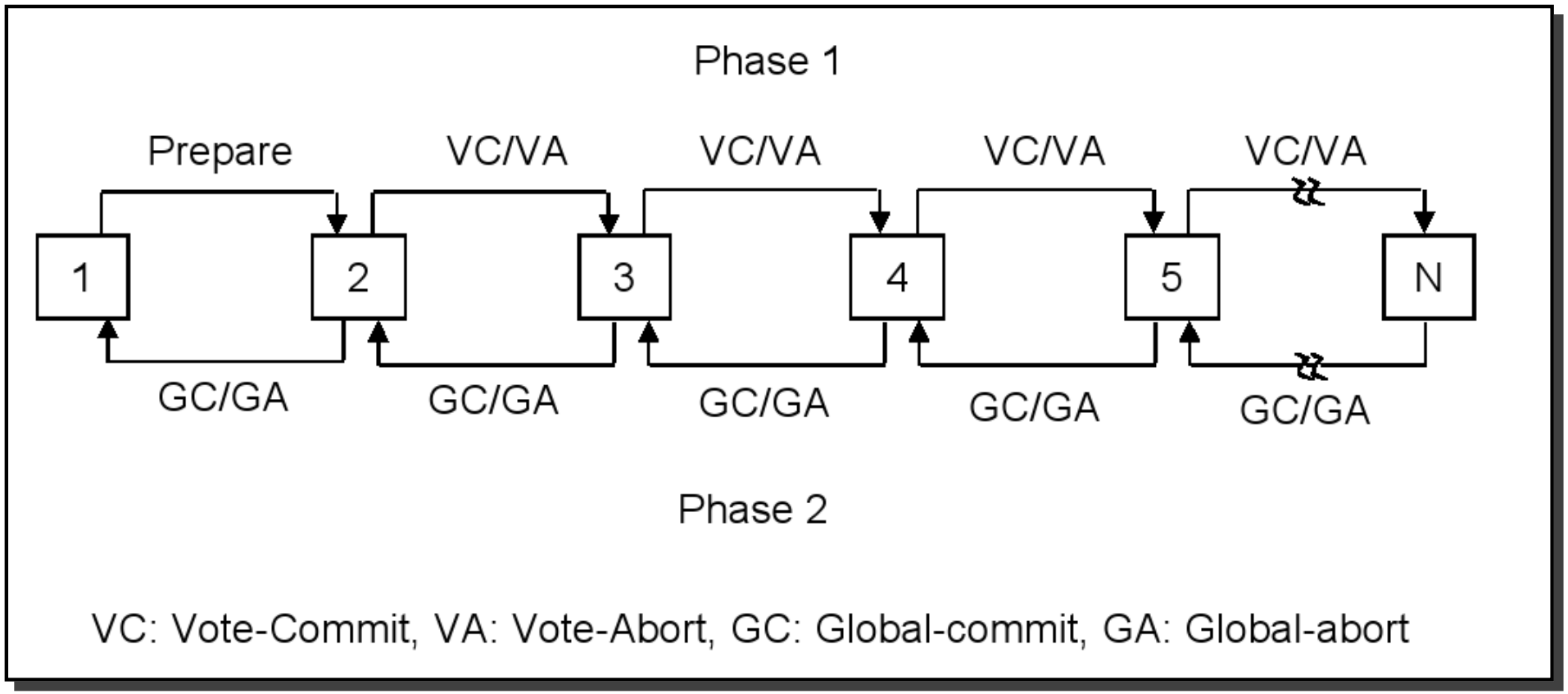


Centralized Two Phase Commit Protocol (2PC) . . .



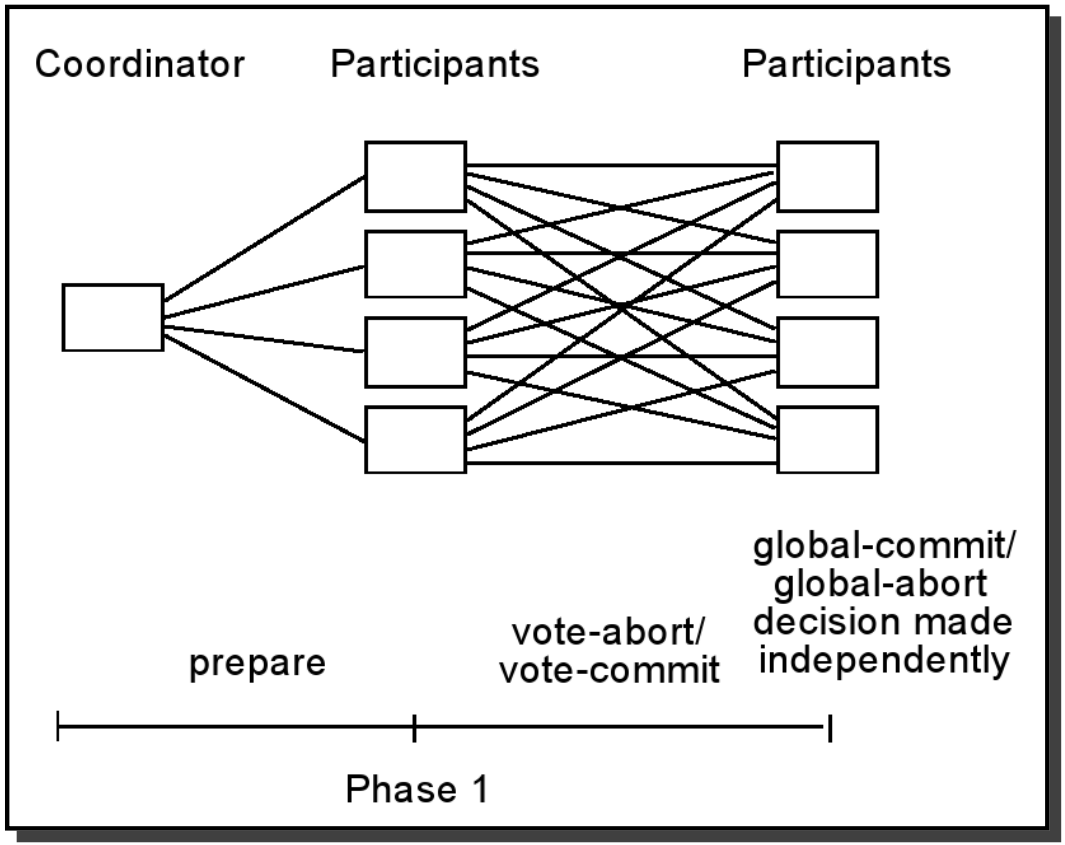
Linear 2PC Protocol

- There is linear ordering between the sites for the purpose of communication
- Minimizes the communication, but low response time as it does not allow any parallelism



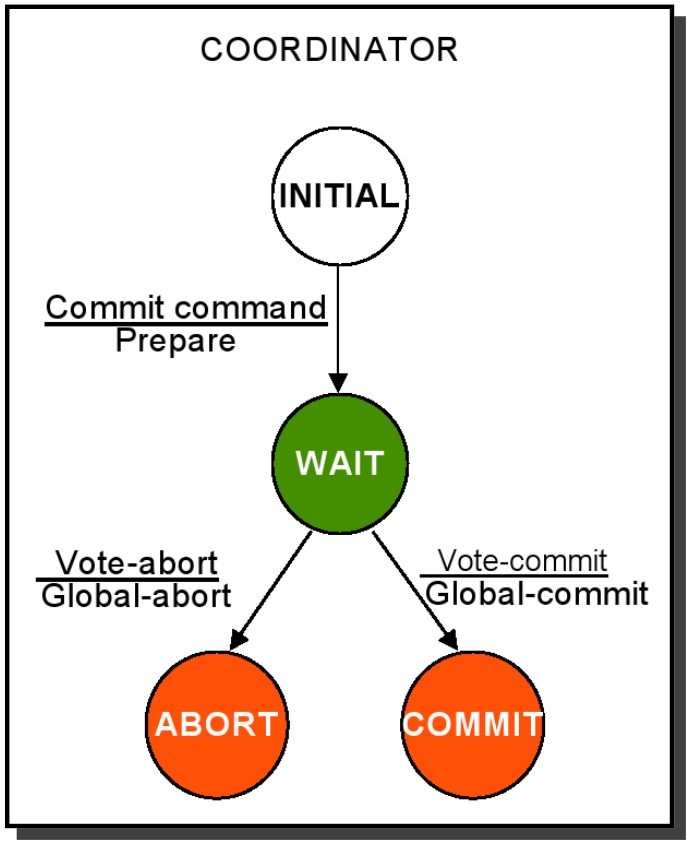
Distributed 2PC Protocol

- Distributed 2PC protocol increases the communication between the nodes
- Phase 2 is not needed, since each participant sends its vote to all other participants (+ the coordinator), thus each participants can derive the global decision



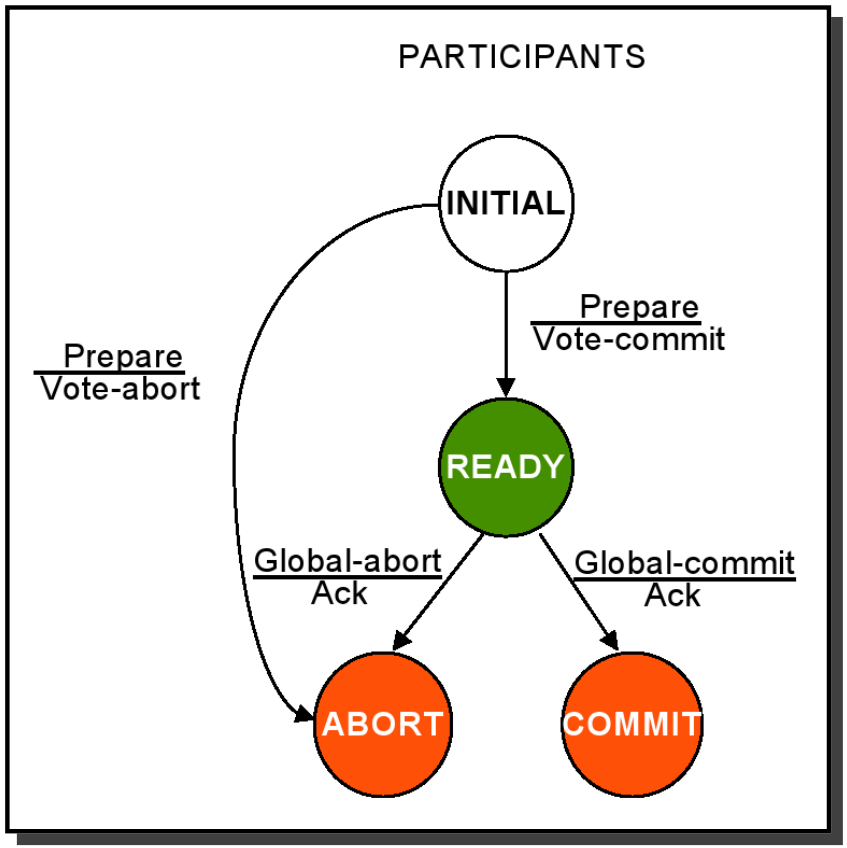
2PC Protocol and Site Failures

- **Site failures** in the 2PC protocol might lead to **timeouts**
- Timeouts are served by **termination protocols**
- We use the state transition diagrams of the 2PC for the analysis
- **Coordinator timeouts:** One of the participants is down. Depending on the state, the coordinator can take the following actions:
  - Timeout in INITIAL
    - Do nothing
  - Timeout in WAIT
    - Coordinator is waiting for local decisions
    - Cannot unilaterally commit
    - Can unilaterally abort and send an appropriate message to all participants
  - Timeout in ABORT or COMMIT
    - Stay blocked and wait for the acks (indefinitely, if the site is down indefinitely)



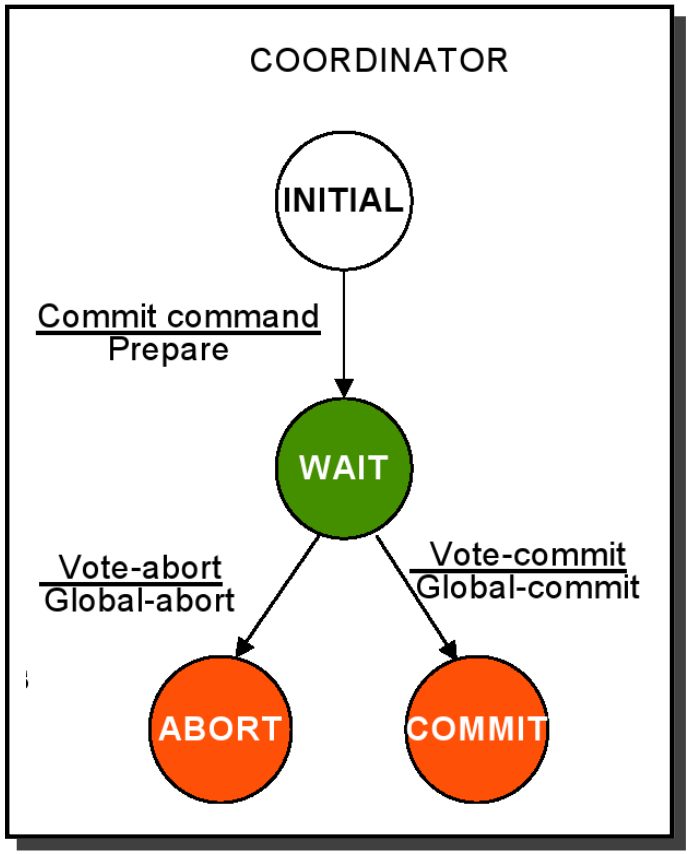
2PC Protocol and Site Failures . . .

- **Participant timeouts:** The coordinator site is down. A participant site is in
  - Timeout in INITIAL
    - Participant waits for “prepare”, thus coordinator must have failed in INITIAL state
    - Participant can unilaterally abort
  - Timeout in READY
    - Participant has voted to commit, but does not know the global decision
    - Participant stays blocked (indefinitely, if the coordinator is permanently down), since participant cannot change its vote or unilaterally decide to commit



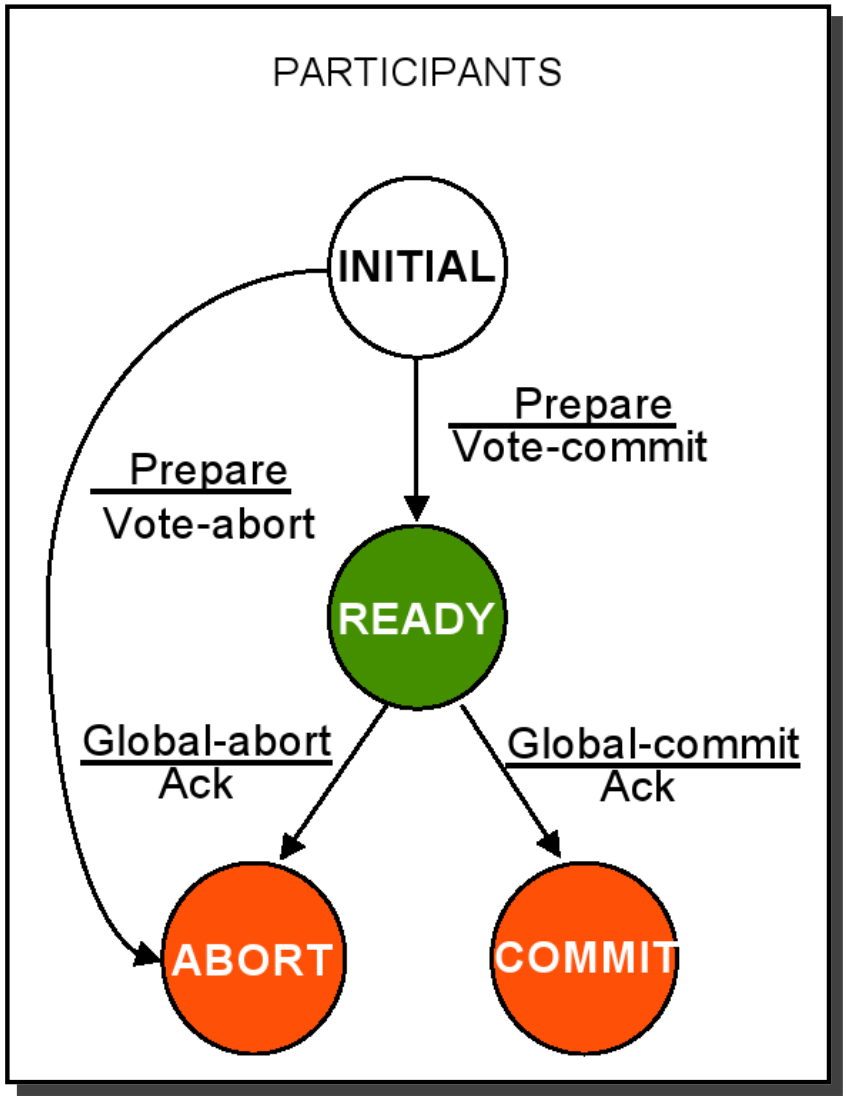
2PC Protocol and Site Failures . . .

- The actions to be taken after a recovery from a failure are specified in the **recovery protocol**
- **Coordinator site failure:** Upon recovery, it takes the following actions:
  - Failure in INITIAL
    - Start the commit process upon recovery (since coordinator did not send anything to the sites)
  - Failure in WAIT
    - Restart the commit process upon recovery (by sending "prepare" again to the participants)
  - Failure in ABORT or COMMIT
    - Nothing special if all the acks have been received from participants
    - Otherwise the termination protocol is involved (re-ask the acks)



2PC Protocol and Site Failures . . .

- **Participant site failure:** The coordinator sites recovers
  - Failure in INITIAL
    - Unilaterally abort upon recovery as the coordinator will eventually timeout since it will not receive the participant's decision due to the failure
  - Failure in READY
    - The coordinator has been informed about the local decision
    - Treat as timeout in READY state and invoke the termination protocol (re-ask the status)
  - Failure in ABORT or COMMIT
    - Nothing special needs to be done



2PC Protocol and Site Failures . . .

- Additional cases
  - Coordinator site fails after writing ”begin_commit” log and before sending ”prepare” command
    - treat it as a failure in WAIT state; send ”prepare” command
  - Participant site fails after writing ”ready” record in log but before ”vote-commit” is sent
    - treat it as failure in READY state
    - alternatively, can send ”vote-commit” upon recovery
  - Participant site fails after writing ”abort” record in log but before ”vote-abort” is sent
    - no need to do anything upon recovery
  - Coordinator site fails after logging its final decision record but before sending its decision to the participants
    - coordinator treats it as a failure in COMMIT or ABORT state
    - participants treat it as timeout in the READY state
  - Participant site fails after writing ”abort” or ”commit” record in log but before acknowledgement is sent
    - participant treats it as failure in COMMIT or ABORT state
    - coordinator will handle it by timeout in COMMIT or ABORT state

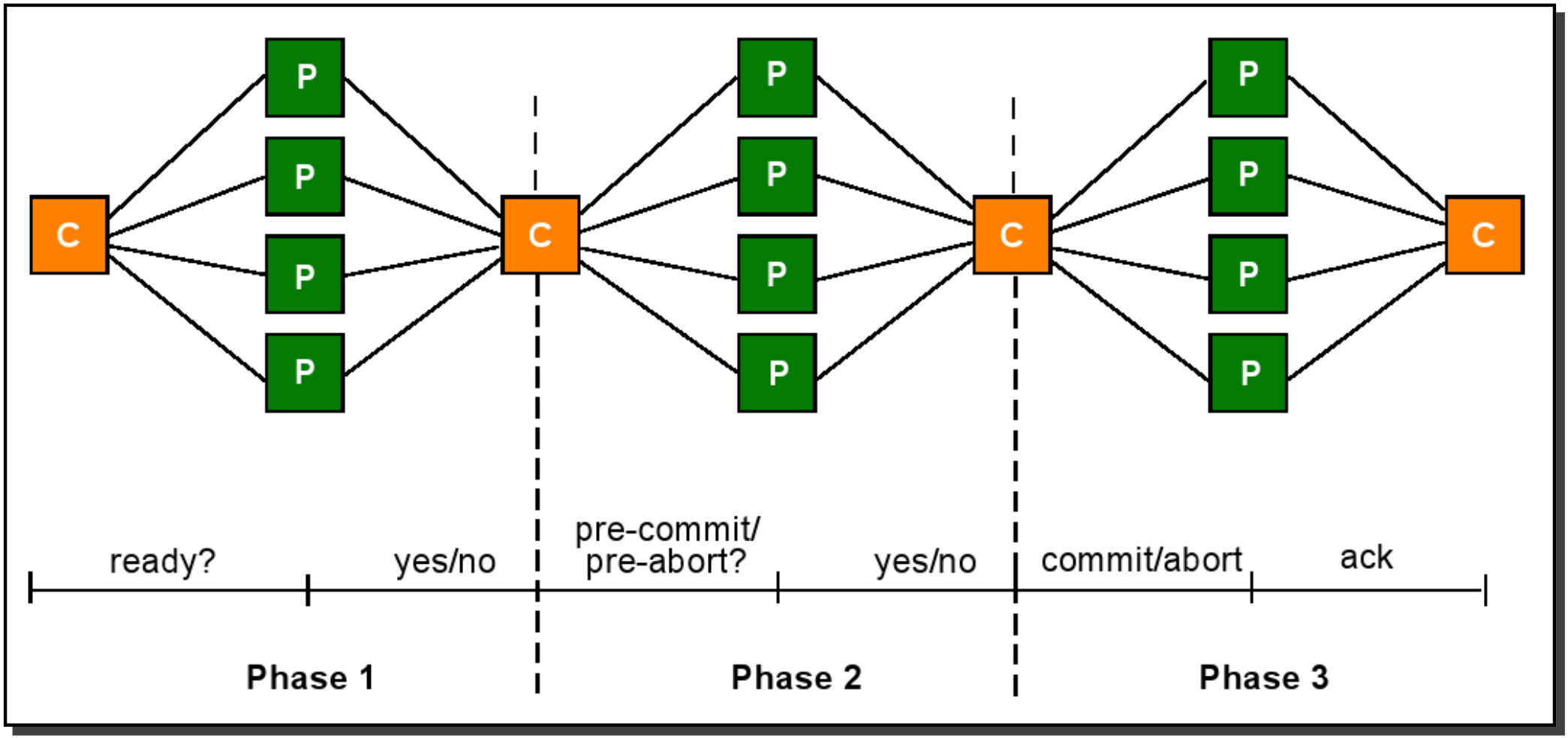
Problems with 2PC Protocol

- A protocol is **non-blocking** if it permits a transaction to terminate at the operational sites without waiting for recovery of the failed site.
  - Significantly improves the response-time of transactions
- 2PC protocol is **blocking**
  - Ready implies that the participant waits for the coordinator
  - If coordinator fails, site is blocked until recovery; independent recovery is not possible
  - The problem is that sites might be in both: commit and abort phases.

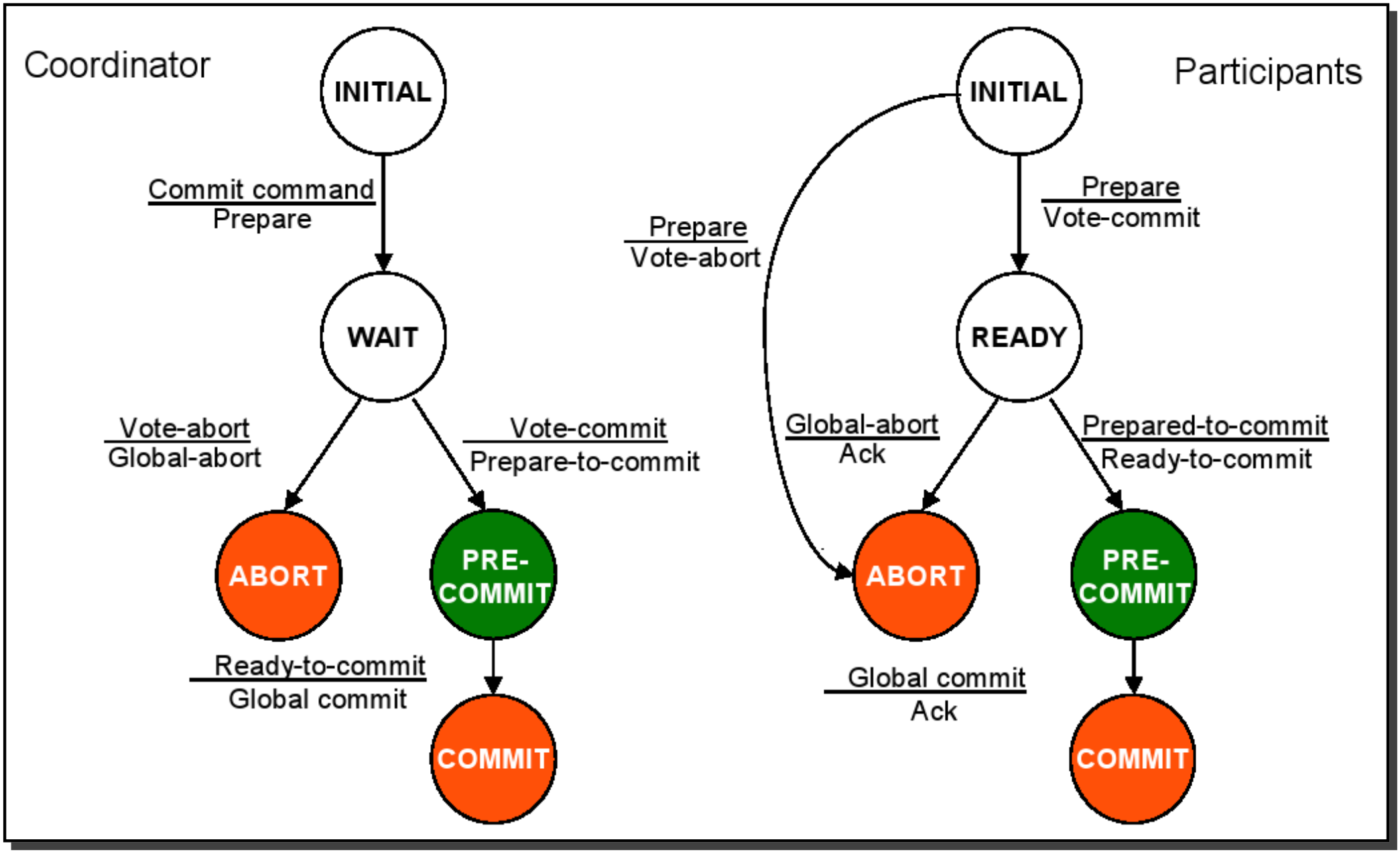
Three Phase Commit Protocol (3PC)

- 3PC is a **non-blocking protocol** when failures are restricted to **single site** failures
- The state transition diagram contains
  - no state which is ”adjacent” to both a commit and an abort state
  - no non-committable state which is ”adjacent” to a commit state
- Adjacent: possible to go from one status to another with a single state transition
- Committable: all sites have voted to commit a transaction (e.g.: COMMIT state)
- Solution: Insert another state between the WAIT (READY) and COMMIT states

Three Phase Commit Protocol (3PC) . . .



Three Phase Commit Protocol (3PC) . . .



Conclusion

- Recovery management enables resilience from certain types of failures and ensures atomicity and durability of transactions
- Local recovery manager (LRM) enables resilience from certain types of failures locally. LRM might employ out-of-place and in-place strategies to deal with updates. In case of the in-place strategy an additional log is used for recovery
- Distributed reliablity protocols are more complicated, in particular the commit, termination, and recovery protocols.
- 2PC protocol first gets participants ready for the transaction (phase 1), and then asks the participants to write the transaction (phase 2). 2PC is a blocking protocol.
- 3PC first gets participants ready for the transaction (phase 1), pre-commits/aborts the transaction (phase 2), and then asks the participants to commit/abort the transaction (phase 3). 3PC is non-blocking.