
Distributed DBMS reliability

Dario Della Monica

These slides are a modified version of the slides provided with the book
Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

The original version of the slides is available at: extras.springer.com

Outline (distributed DB)

- Introduction (Ch. 1) *
- Distributed Database Design (Ch. 3) *
- Distributed Query Processing (Ch. 6-8) *
- Distributed Transaction Management (Ch. 10-12) *
 - Introduction to transaction management (Ch. 10) *
 - Distributed Concurrency Control (Ch. 11) *
 - **Distributed DBMS Reliability (Ch. 12) ***

* Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Outline (today)

- Distributed DBMS Reliability (Ch. 12) ^{*}
 - Introduction and local reliability protocols
 - Distributed reliability protocols
 - ◆ Two-phase commit (2PC) protocol

^{*} Özsu and Valduriez, *Principles of Distributed Database Systems* (3rd Ed.), 2011

Reliability

Problem:

How to maintain

atomicity

durability

properties of transactions

Fundamental Definitions

- Reliability
 - A measure of success with which a system conforms to some authoritative specification of its behavior
- Availability
 - The fraction of the time that a system meets its specification
- Failure
 - The deviation of a system from the behavior that is described in its specification

Types of Failures

- Transaction failures
 - Transaction aborts (unilaterally or due to deadlock)
- System (site) failures
 - Failure of processor, main memory, power supply, ...
 - Main memory contents are lost, but secondary storage contents are safe
 - Partial (some sites) vs. total (all sites) failure
- Media failures
 - Failure of secondary storage devices such that the stored data is lost
 - Head crash/controller failure (?)
 - Permanent data loss (secondary, resilient, stable memory – hard disk)
- Communication failures
 - Lost/undeliverable messages
 - Network partitioning

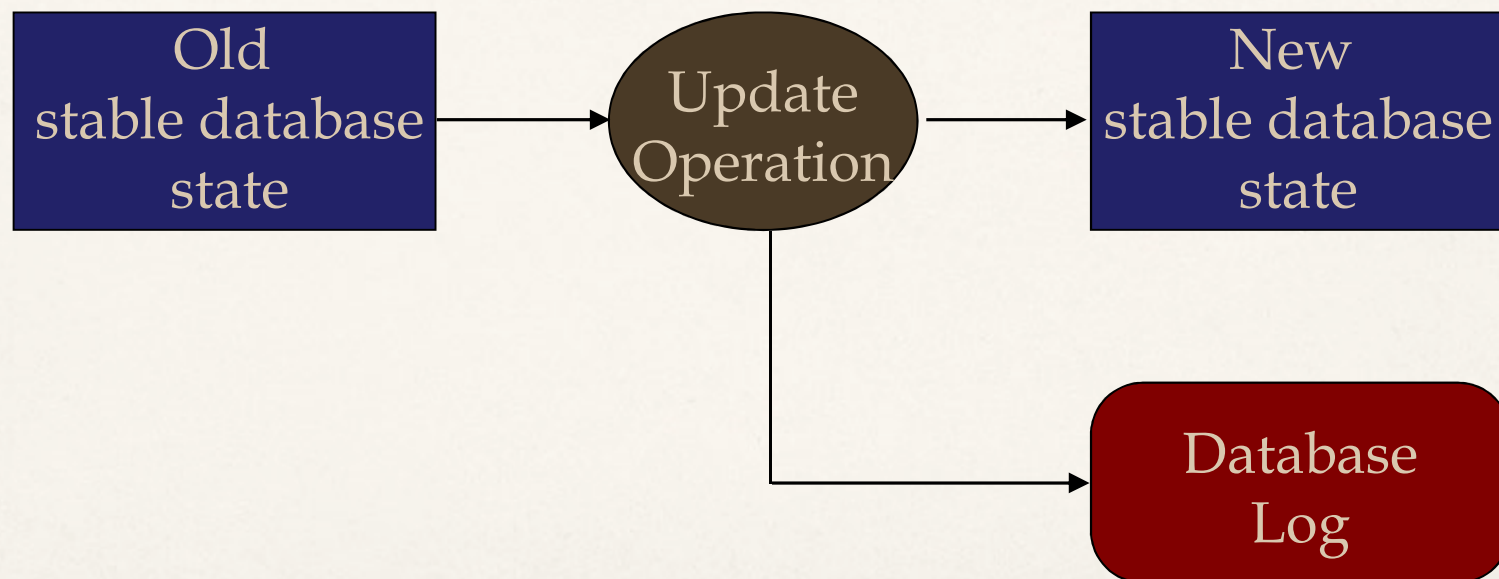
Update Strategies

- In-place update
 - Each update causes a change in one or more data values in the database
 - More efficient, more difficult to undo
- Out-of-place update
 - Each update causes the new value(s) of data item(s) to be stored separately from the old value(s)
 - Less efficient, easy to undo

In-Place Update Recovery Information

Database Log

Every action of a transaction must not only perform the action, but must also write a *log* record to an append-only file.



Logging

The log contains information used by the recovery process to restore the consistency of a system. This information may include

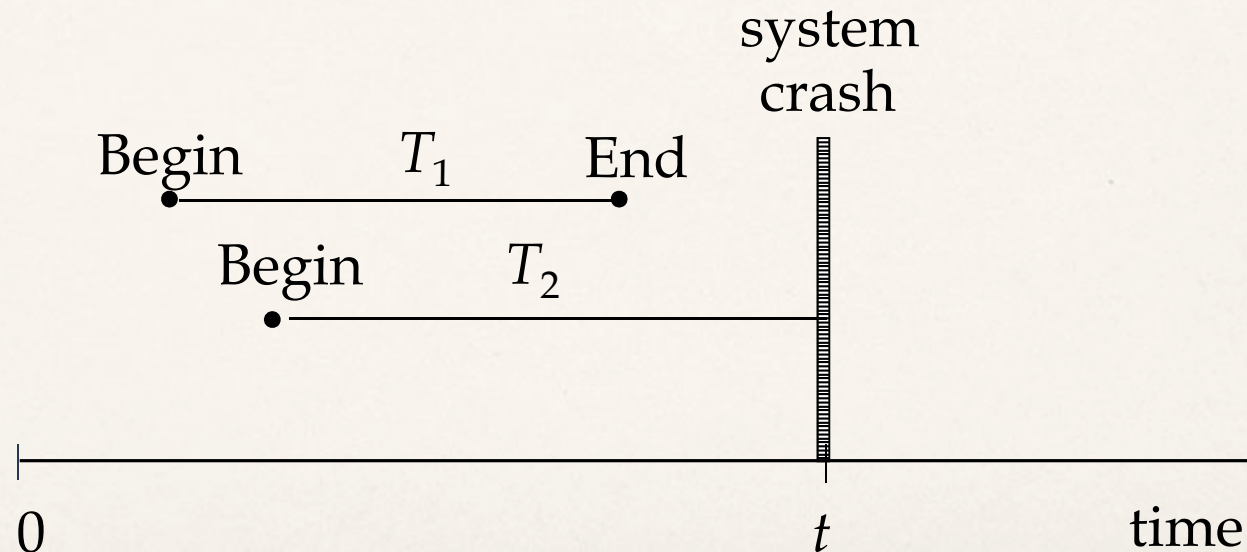
- transaction identifier
- type of operation (action)
- items accessed by the transaction to perform the action
- old value (state) of item (**before image**)
- new value (state) of item (**after image**)

...

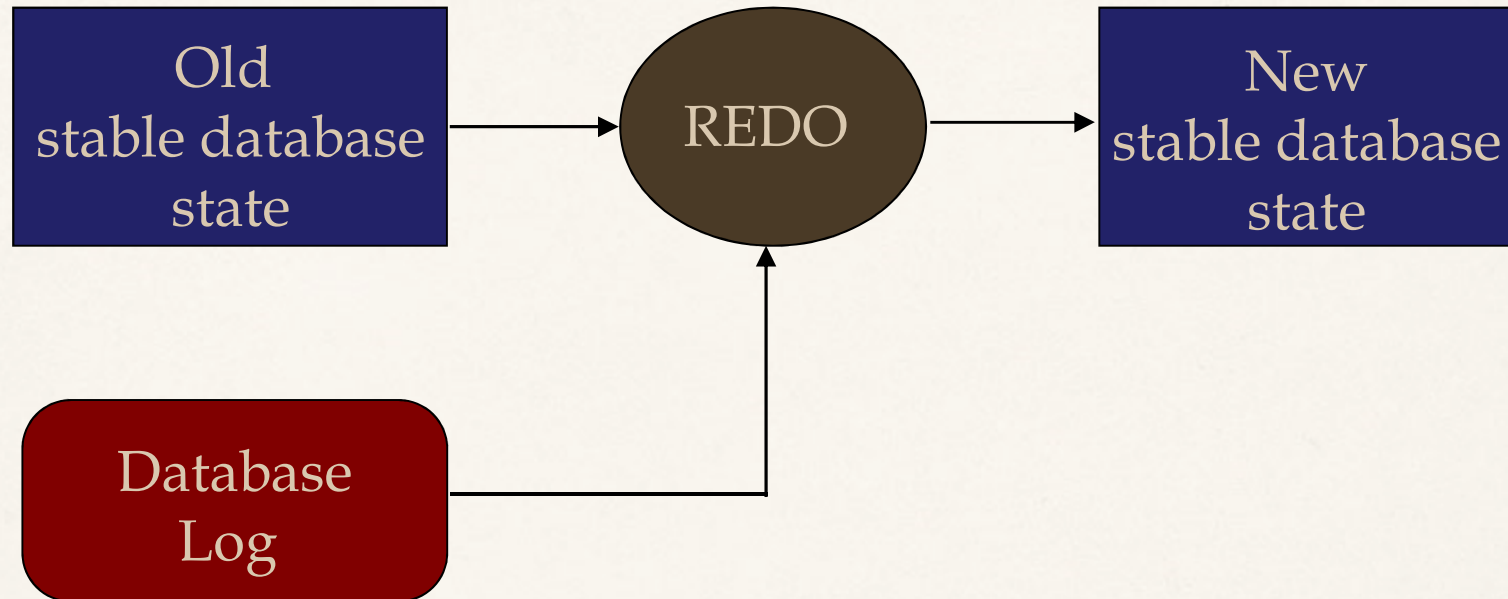
Why Logging?

Upon recovery:

- all of T_1 's effects should be reflected in the database (REDO if necessary due to a failure)
- none of T_2 's effects should be reflected in the database (UNDO if necessary)

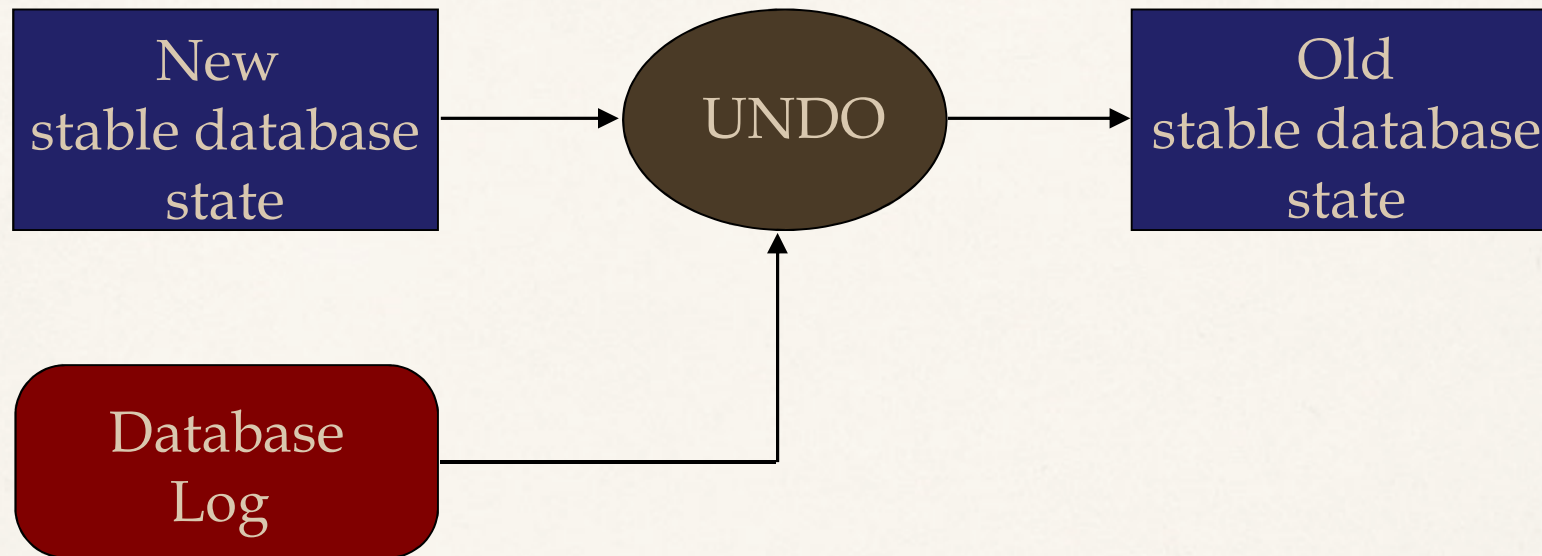


REDO Protocol



- REDO'ing an action means performing it again
- The REDO operation uses the log information
- REDO is needed when effects of a committed transaction were not stored yet in secondary (stable, resilient) memory
 - sometimes for efficiency reasons storing information to disk (secondary memory) is done at a later time

UNDO Protocol



- UNDO'ing an action means to restore the object to its before image
- The UNDO operation uses the log information
- UNDO is needed when effects of a transaction are stored in secondary (stable, resilient) memory and then an abort occurs
 - sometimes to free main memory, information is stored to disk (secondary memory) before commit

When to Write Log Records Into Stable Store

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

Write–Ahead Log Protocol

- Notice:
 - If a system crashes before a transaction is committed, then all the operations must be undone. Only need the before images (*undo portion* of the log)
 - Once a transaction is committed, some of its actions might have to be redone. Need the after images (*redo portion* of the log)
- WAL protocol :
 - ① Before a stable database is updated, the undo portion of the log should be written to the stable log
 - ② When a transaction commits, the redo portion of the log must be written to stable log prior to the updating of the stable database.

Execution of Commands

Commands to consider:

begin_transaction

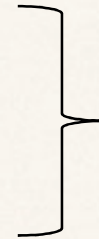
read

write

abort

commit

recover



Independent of execution
strategy for LRM

Execution Strategies

- Dependent upon
 - Can the buffer manager (**BM**) decide to write some of the buffer pages being accessed by a transaction into stable storage or does it wait for LRM to instruct it?
 - ♦ fix/no-fix decision (*fix* means BM **cannot** store the data into disk before commit)
 - (*no-fix* means BM **can** store data to disk before commit)
 - Does the LRM force the buffer manager to write certain buffer pages into stable database at the end of a transaction's execution?
 - ♦ flush/no-flush decision (*flush* means BM **cannot** wait; it must store data into disk at commit)
 - (*no-flush* means BM **can** wait; it can store data into disk at a later time)
- Possible execution strategies:
 - no-fix/no-flush
 - no-fix/flush
 - fix/no-flush
 - fix/flush

No-Fix/No-Flush

- Abort
 - Buffer manager may have written some of the updated pages into stable database (second memory, disk)
 - LRM performs **transaction undo**
- Commit
 - LRM writes an “end_of_transaction” record into the log
 - Data not necessarily written into disk
- Recover
 - For those transactions that have both a “begin_transaction” and an “end_of_transaction” record in the log, a **redo** is initiated by LRM
 - For those transactions that only have a “begin_transaction” in the log, an **undo** is executed by LRM

No-Fix/Flush

- Abort
 - Buffer manager may have written some of the updated pages into stable database (second memory, disk)
 - LRM performs transaction undo
- Commit
 - LRM issues a `flush` command to the buffer manager for all updated pages
 - ◆ i.e., data is stored into disk at time of commit
 - LRM writes an “`end_of_transaction`” record into the log
- Recover
 - No need to perform redo
 - Perform undo

Fix/No-Flush

- Abort
 - None of the updated pages have been written into stable database
 - Release the `fixed` pages
- Commit
 - LRM writes an “`end_of_transaction`” record into the log
 - Data not necessarily written into disk
 - LRM sends an `unfix` command to the buffer manager for all pages that were previously `fixed`
- Recover
 - Perform redo
 - No need to perform undo

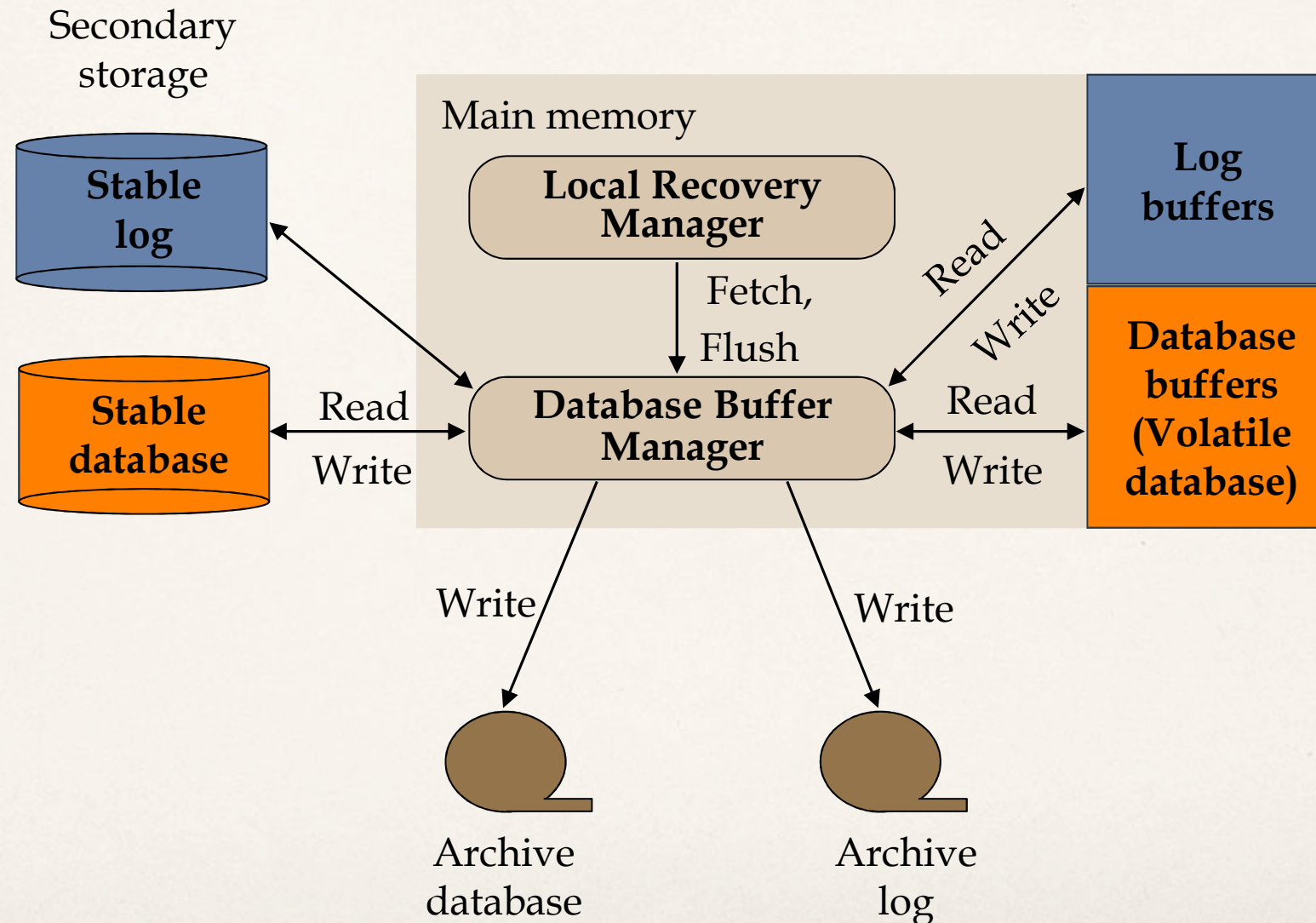
Fix/Flush

- Abort
 - None of the updated pages have been written into stable database
 - Release the `fixed` pages
- Commit (the following have to be done atomically)
 - LRM issues a `flush` command to the buffer manager for all updated pages
 - ◆ i.e., data is stored into disk at time of commit
 - LRM sends an `unfix` command to the buffer manager for all pages that were previously `fixed`
 - LRM writes an “`end_of_transaction`” record into the log
- Recover
 - No need to do anything

Checkpoints

- Simplifies the task of determining actions (of transactions) that need to be undone or redone when a failure occurs
 - Avoid scanning the whole log
- A checkpoint identify a consistent state of the DB
- Steps to create a checkpoint:
 - ① Write a begin_checkpoint record into the log
 - ② Collect the checkpoint data into the stable storage (log and actual DB data)
 - During this phase stop accepting new transactions, complete all currently active ones
 - ③ Write an end_checkpoint record into the log

Media Failures – Full Architecture



Distributed Reliability Protocols

- Commit protocols
 - How to execute commit command for distributed transactions
 - Issue: how to ensure atomicity and durability?
- Termination protocols
 - If a failure occurs, how the remaining operational sites behave
 - *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 the sites where the failure occurred behave after they are back on
 - *Independent* : a failed site can determine the outcome of a transaction without having to obtain remote information.
- Independent recovery \Rightarrow non-blocking termination

Two-Phase Commit (2PC)

- **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

Phase 1 : The coordinator gets the participants ready to commit or abort and collects their reply

Phase 2 : The coordinator decides global-abort/ global-commit depending on participants' replies, communicate the decision to them, and waits for ack's

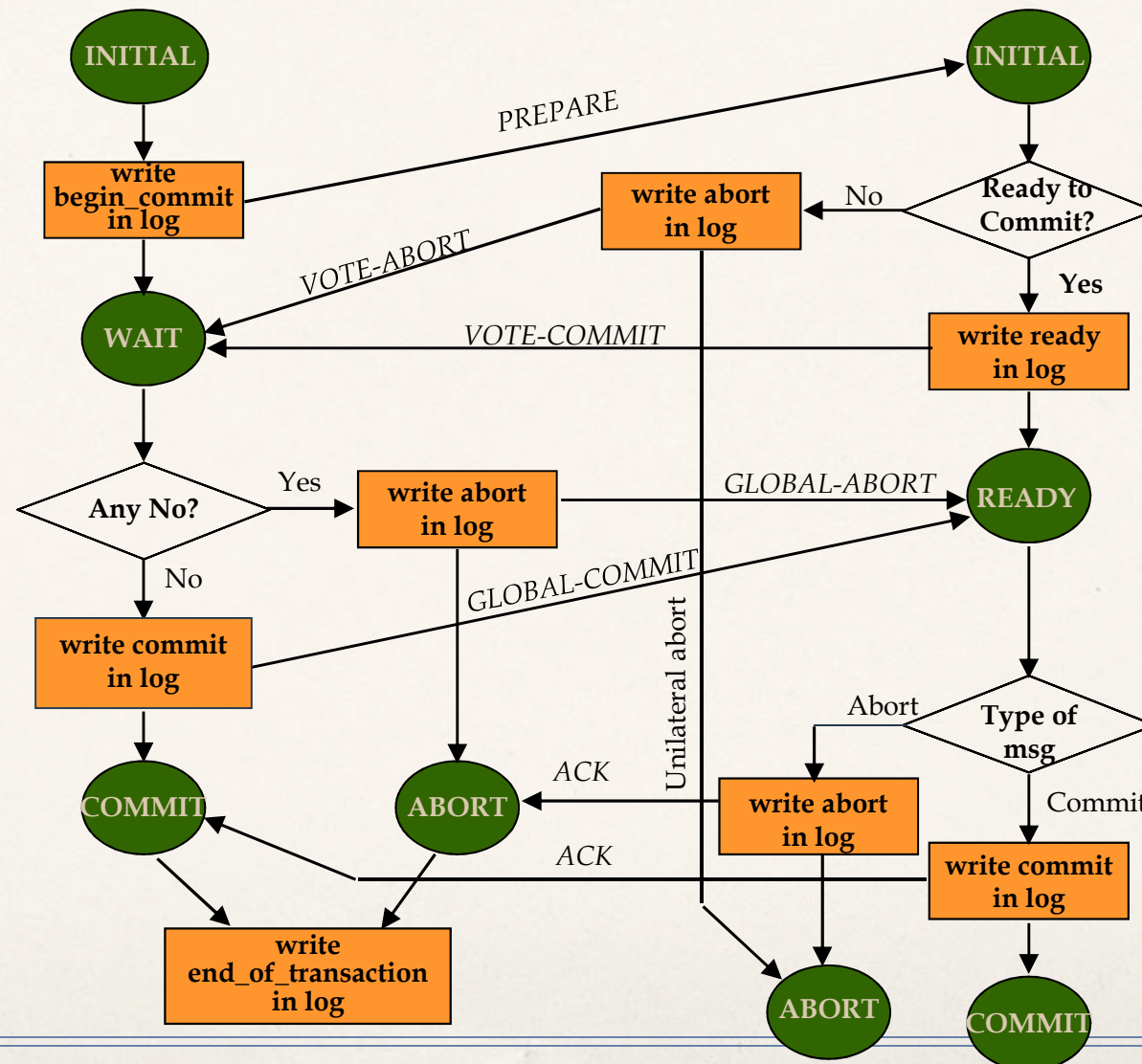
Global Commit Rule:

- The coordinator aborts a transaction if and only if at least one participant votes to abort it
 - Equivalently: The coordinator commits a transaction if and only if all of the participants vote to commit it

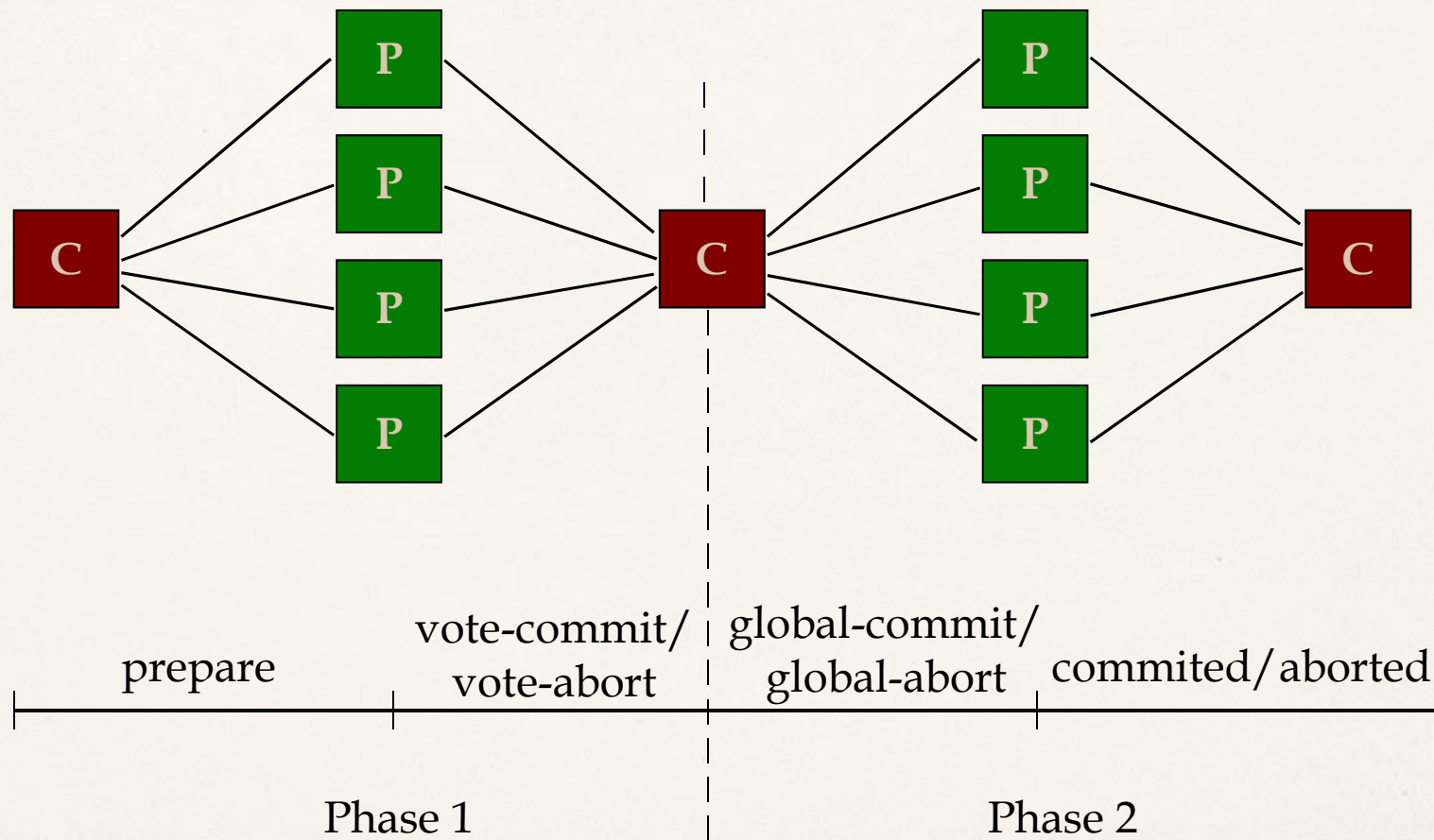
2PC Protocol Actions

Coordinator

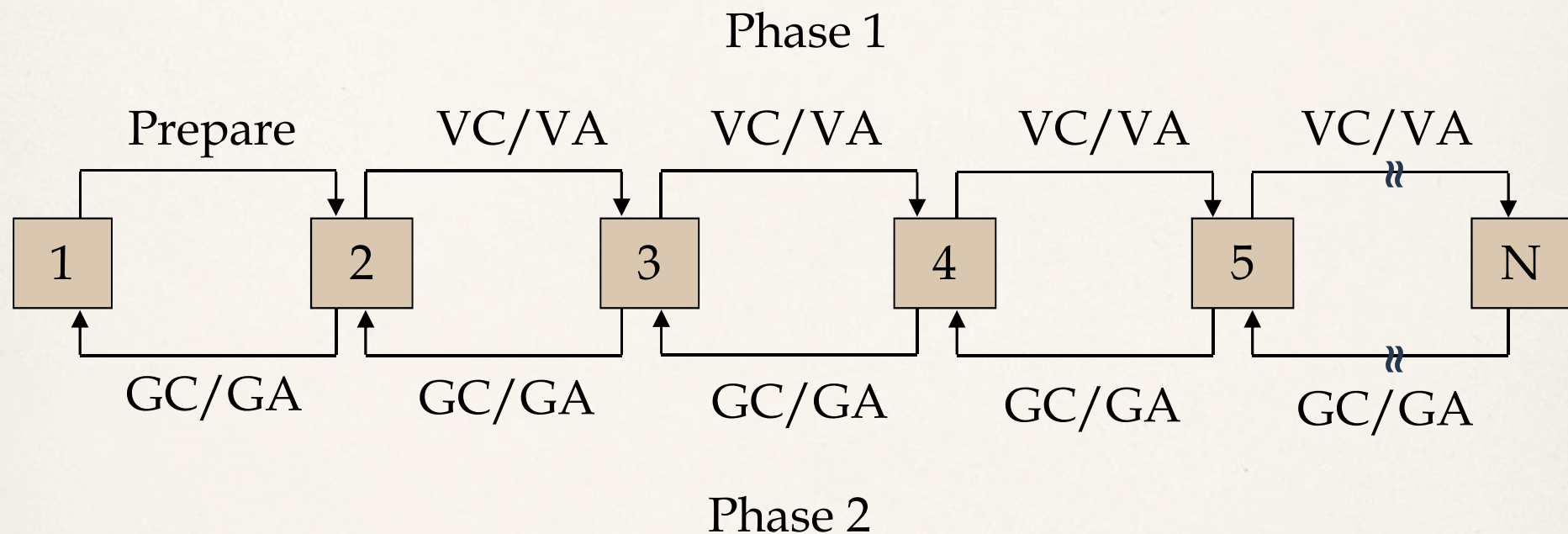
Participant



Centralized 2PC



Linear 2PC



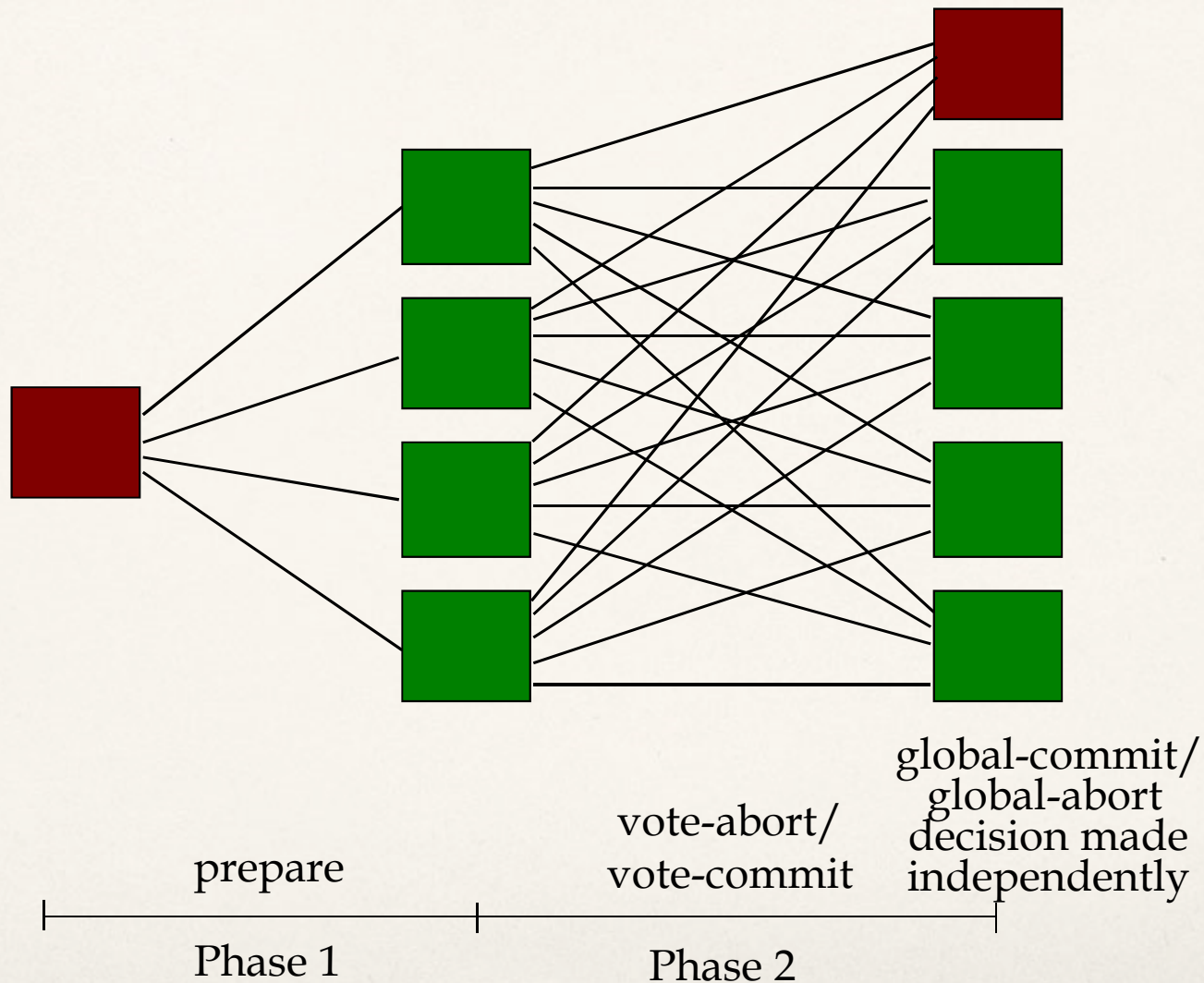
VC: Vote-Commit, VA: Vote-Abort, GC: Global-commit, GA: Global-abort

Distributed 2PC

Coordinator

Participants

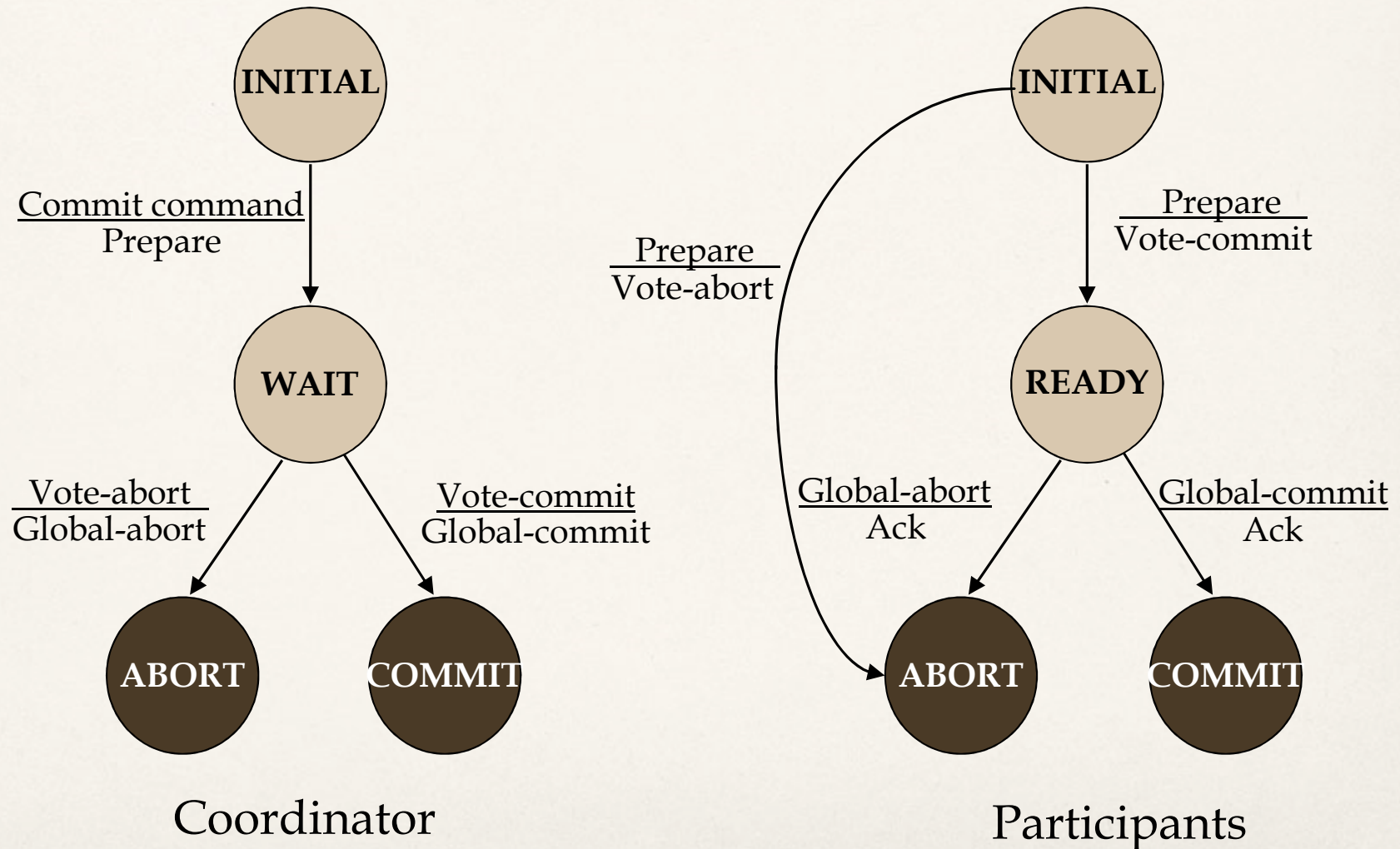
Participants



Variations of 2PC

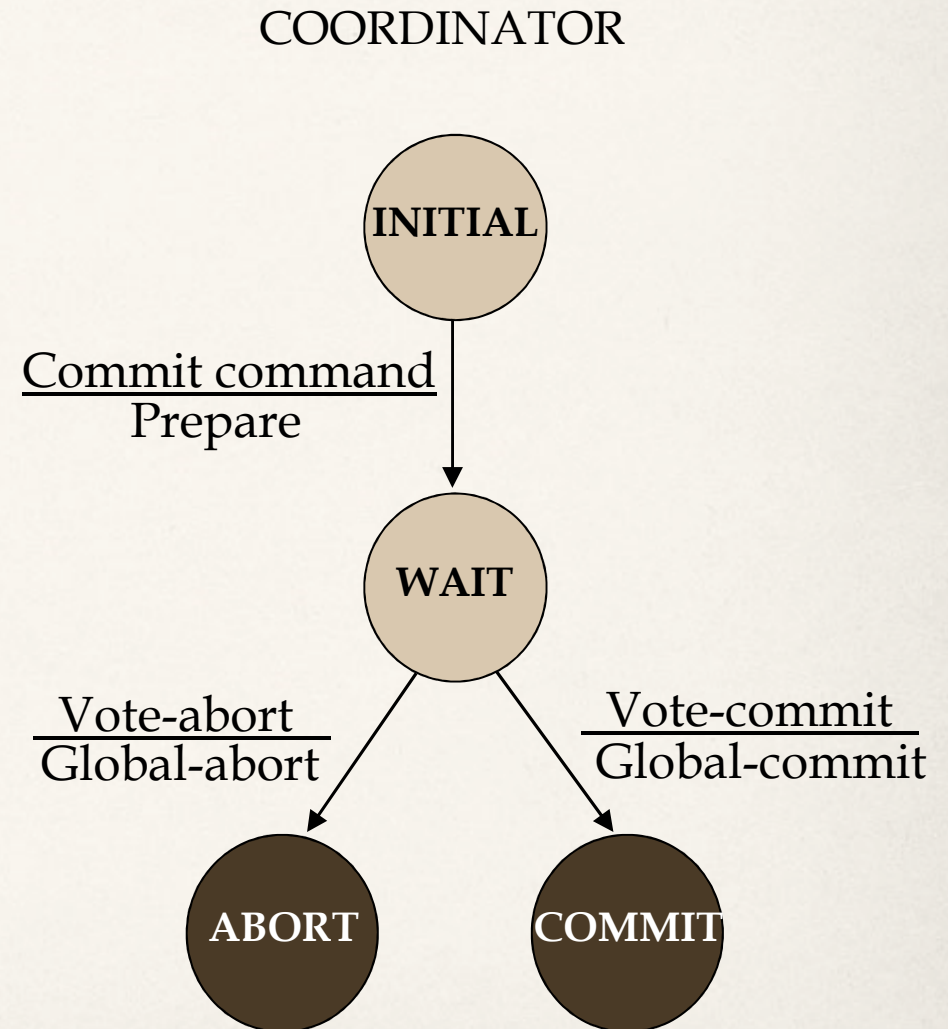
- Presumed abort 2PC and presumed commit 2PC
- Coordinator and participant may assume global-abort or global-commit if they do not get communication
 - Reduced communication

State Transitions in 2PC



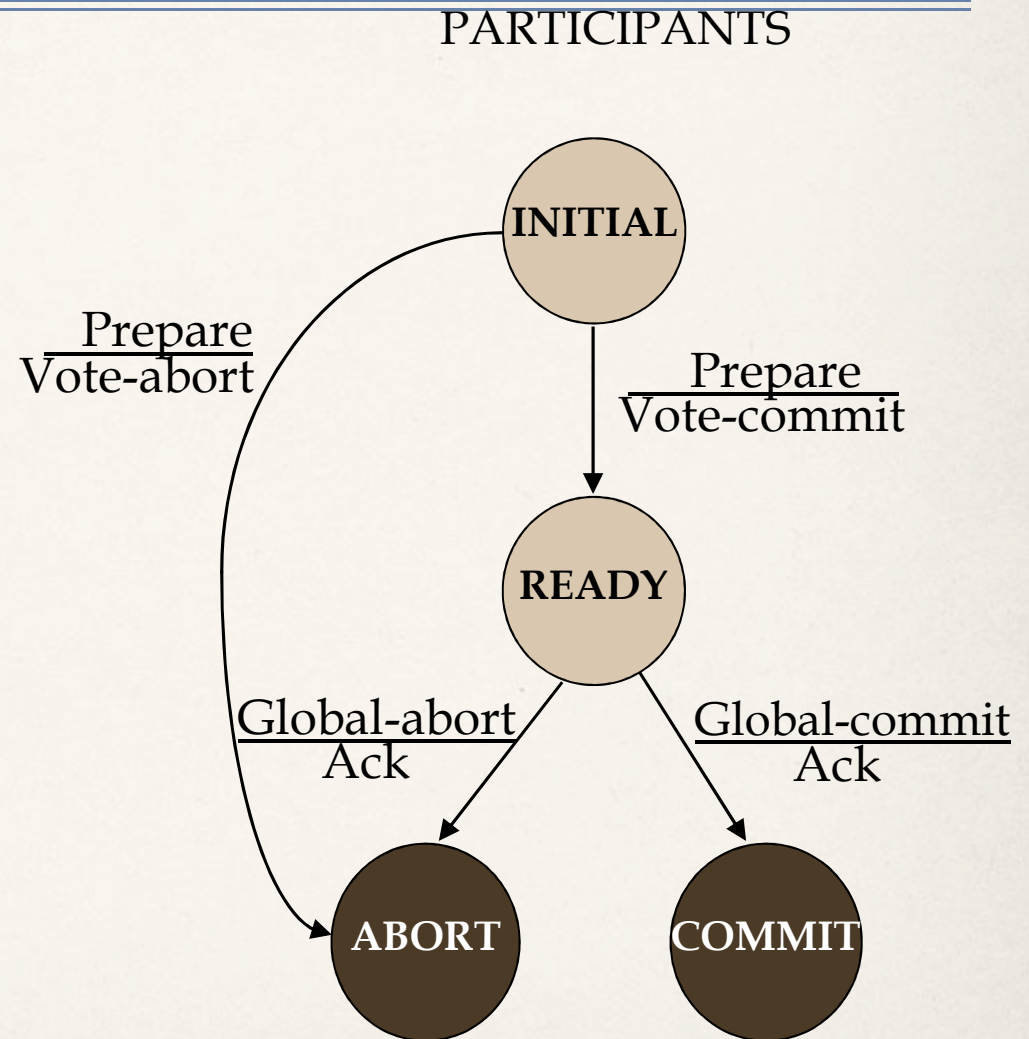
Site Failures - 2PC Termination

- Timeout in WAIT
 - Cannot unilaterally commit
 - Can unilaterally abort
- Timeout in ABORT or COMMIT
 - Stay blocked and wait for the acks
 - Repeatedly send “global-commit” or “global-abort” to unresponsive participants



Site Failures - 2PC Termination

- Timeout in INITIAL
 - Coordinator must have failed in INITIAL state
 - Unilaterally abort
- Timeout in READY
 - Stay blocked
 - Repeatedly send “vote-commit” to coordinator
- If participants can communicate, they can resolve blocked situations. Assume P_i timed out in READY and it asks to P_j
 - P_j in INITIAL: P_j abort
 - P_j in READY: nothing can be done
 - P_j in ABORT/COMMIT: P_j send “vote-commit” / “vote-abort” to P_i

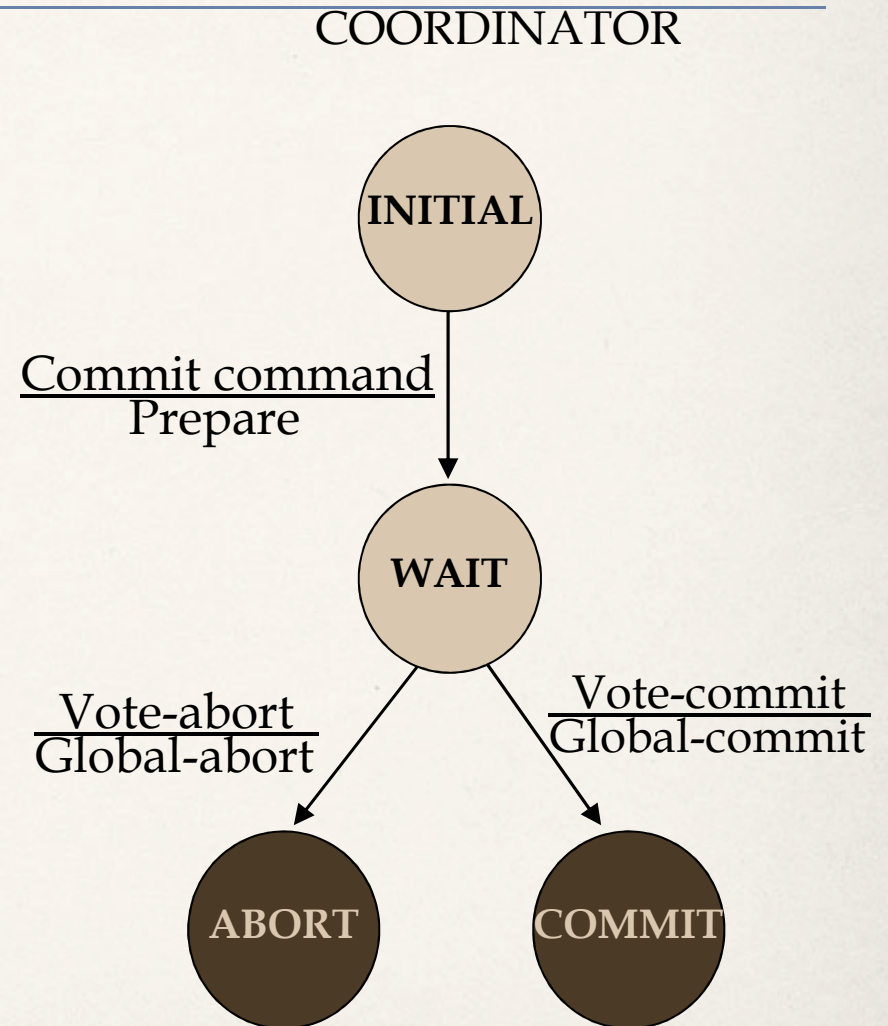


Re-election of the coordinator

- If participants can communicate ...
- ... and all of them know that the coordinator site is the only failing one
- then another coordinator is elected and the protocol is re-started
 - Election by ordering participants or by any voting procedure
- Does not work if a participant site fails besides the coordinator. Indeed:
 - Participant receive communication from coordinator
 - Participant terminate transaction accordingly
 - Participant and coordinator sites both fail
 - A new execution of the protocol among the remaining participants through re-election of coordinator might lead to a different decision
- 2PC is a blocking protocol

Site Failures - 2PC Recovery

- Failure in INITIAL
 - Start the commit process upon recovery
- Failure in WAIT
 - Restart the commit process upon recovery
- Failure in ABORT/COMMIT
 - Nothing special if all the acks have been received
 - Otherwise invoke the termination protocol for timeout in ABORT/COMMIT



Site Failures - 2PC Recovery

- Failure in INITIAL
 - Unilaterally abort upon recovery
- Failure in READY
 - The coordinator has been informed about the local decision
 - Treat as timeout in READY state and invoke the termination protocol
- Failure in ABORT or COMMIT
 - Nothing special needs to be done

PARTICIPANTS



2PC Recovery Protocols – Additional Cases

Arise due to non-atomicity of log and message send actions

- Coordinator site fails after writing “begin_commit” log and before sending “prepare” command
 - treat it as a failure in WAIT state; invoke recovery protocol from WAIT (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
 - invoke recovery protocol from READY
- Participant site fails after writing “abort” record in log but before “vote-abort” is sent
 - no need to do anything upon recovery

2PC Recovery Protocols – Additional Cases (cont'd)

- 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
 - participants treat it as failure in COMMIT or ABORT state
 - ◆ send ACK message upon request
 - coordinator will handle it by timeout in COMMIT or ABORT state

Problem With 2PC

- Blocking
 - “Ready” state implies that the participant waits for the coordinator
 - If coordinator fails, site is blocked until recovery
 - Blocking reduces availability
- Independent recovery is not possible
- However, it is known that:
 - Independent recovery protocols exist only for single site failures; no independent recovery protocol exists which is resilient to multiple-site failures.
- 3PC is non-blocking (for (single) site failures)
- Communication line failures (network partitioning) are more problematic
 - No non-blocking protocol exists

More Problematic Failure Types

- We only considered *failures of omission*
 - A message is not received, a site is unresponsive
- *Failures of commissions*
 - Implementation errors (system does not work as expected): incorrect messages
 - Malicious behaviors: a participant pretends to be the coordinator
 - Addressed using *byzantine agreement*