Reliability

Problem:

How to maintain

atomicity

durability

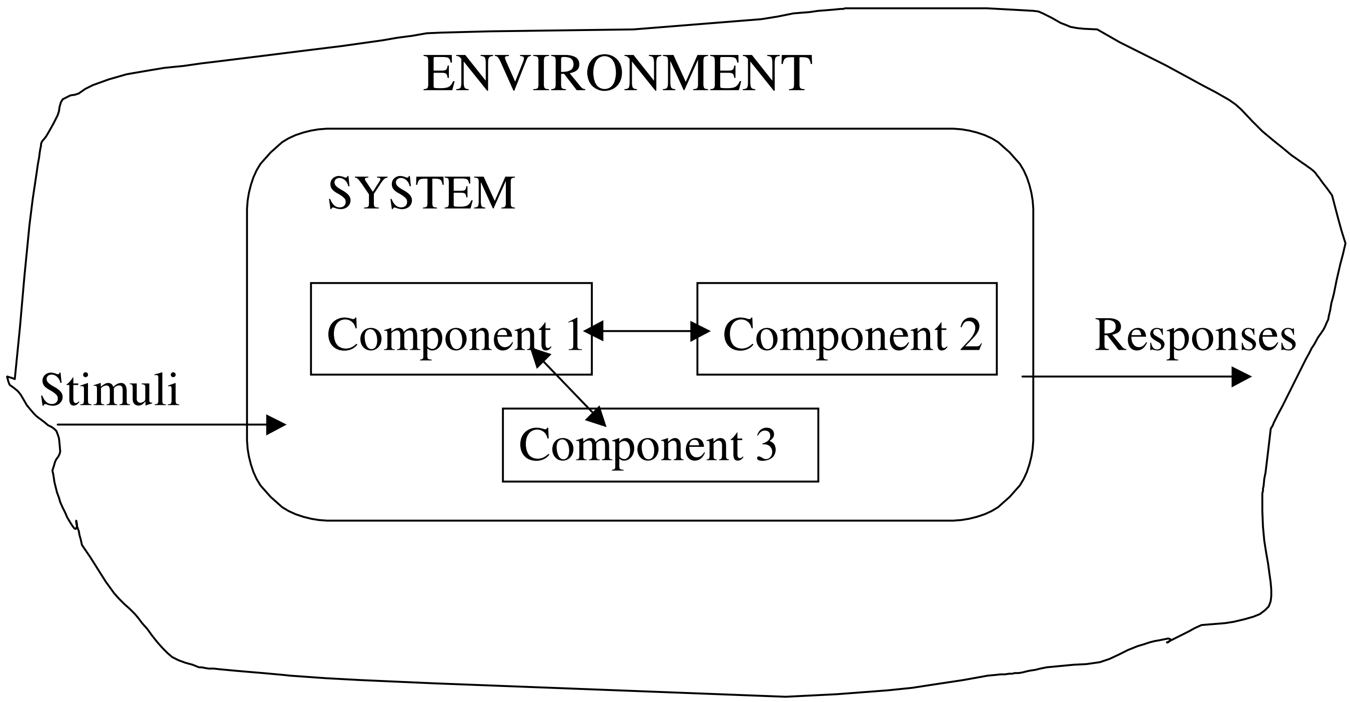
properties of transactions

Fundamental Definitions

- Reliability
  - A measure of success with which a system conforms to some authoritative specification of its behavior.
  - Probability that the system has not experienced any failures within a given time period.
  - Typically used to describe systems that cannot be repaired or where the continuous operation of the system is critical.

- Availability
  - The fraction of the time that a system meets its specification.
  - The probability that the system is operational at a given time *t*.

Basic System Concepts



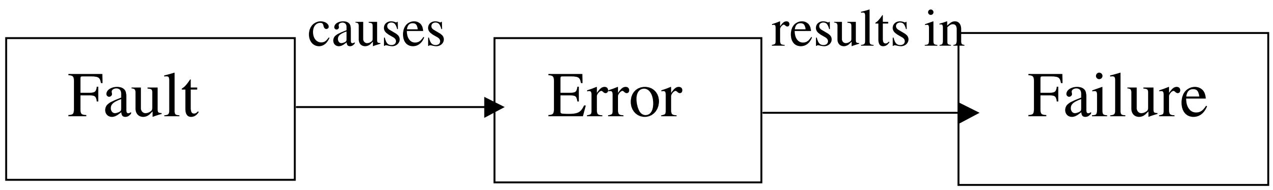
External state

Internal state

Fundamental Definitions

- Failure
  - The deviation of a system from the behavior that is described in its specification.
- Erroneous state
  - The internal state of a system such that there exist circumstances in which further processing, by the normal algorithms of the system, will lead to a failure which is not attributed to a subsequent fault.
- Error
  - The part of the state which is incorrect.
- Fault
  - An error in the internal states of the components of a system or in the design of a system.

Faults to Failures

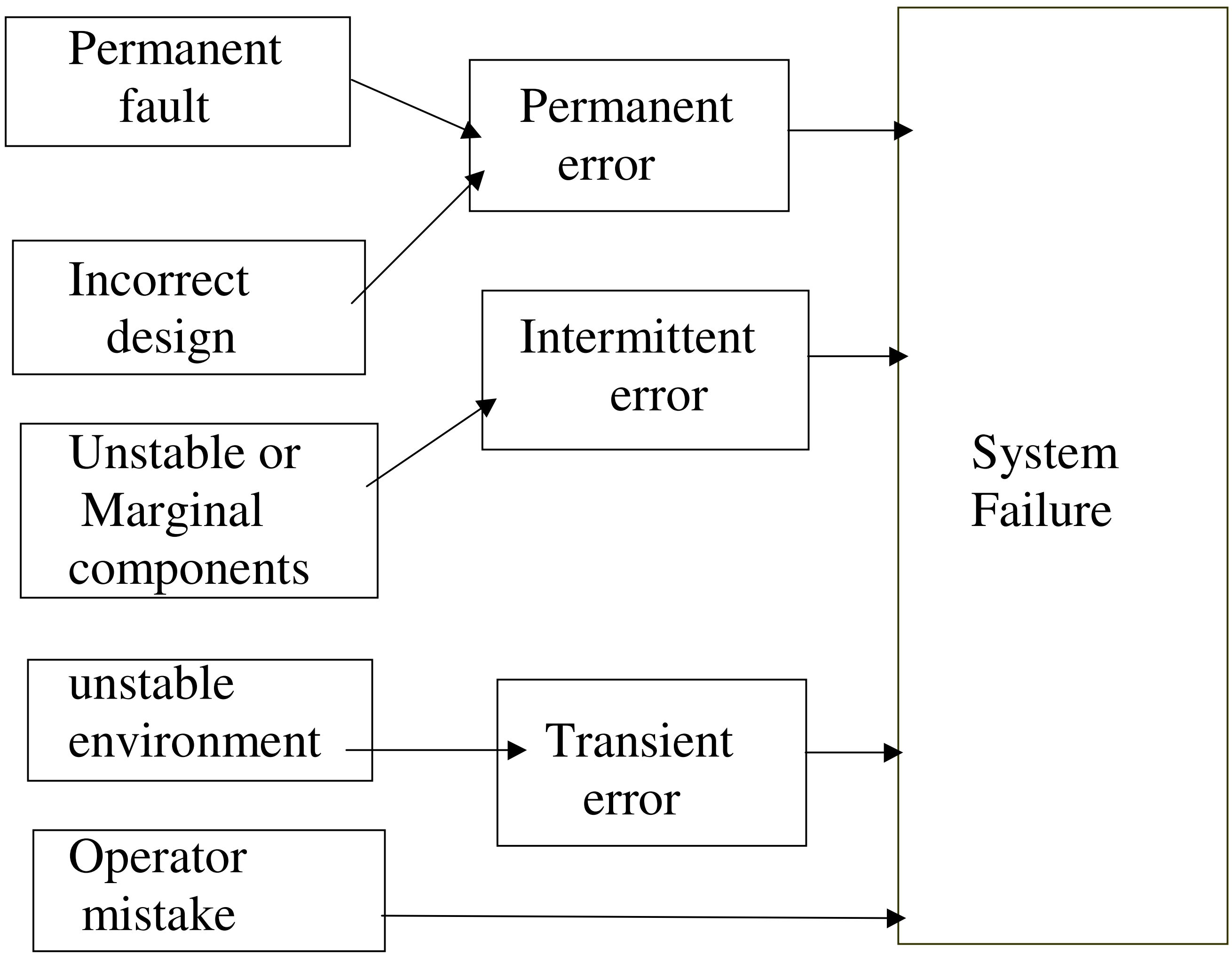


Types of Faults

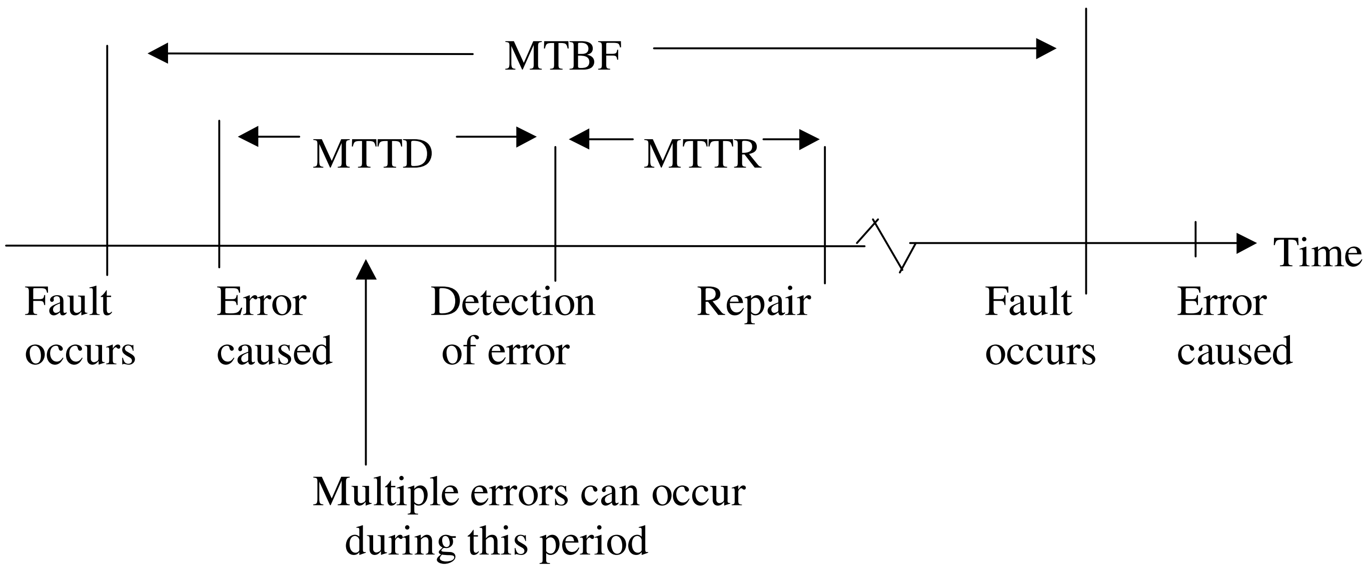
- Hard faults
  - Permanent
  - Resulting failures are called hard failures

- Soft faults
  - Transient or intermittent
  - Account for more than 90% of all failures
  - Resulting failures are called soft failures

Fault Classification



Failures



MTBF – mean time between failure
MTTD – mean time to detect
MTTR – mean time to repair

Fault Tolerance Measures

Reliability

$R(t)$ = Pr{0 failures in time [0, t] | no failures at t=0}

If occurrence of failures is Poisson

$R(t)$ = Pr{0 failures in time [0, t]}

Then

Pr(k failures in time [0,t]) =

$$\frac{e^{-m(t)}[m(t)]^k}{k!}$$

Where $m(t)$ is known as the *hazard function* which gives the time-dependent failure rate of the component and is defined as

$$m(t) = \int_0^t z(x)dx$$

Fault-Tolerance Measures

Reliability

The mean number of failures in time [0,*t*] can be computed as

$$E[k] = \sum_{k=0}^{\infty} k \frac{e-m^{(t)}[m(t)]k}{k!} = m(t)$$

And the variance can be computed as

$$Var[k] = E[k^2] - (E[k])^2 = m(t)$$

Thus, reliability of a single component is

$$R(t) = e^{-m(t)}$$

And of a system consisting of *n* non-redundant components as

$$R_{sys}(t) = \prod_{i=1}^{n} R_i(t)$$

Fault-Tolerance Measures

Availability

A(t) = Pr{system is operational at time t}

Assume

- Poisson failures with rate λ
- Repair time is exponentially distributed with mean $1/\mu$

Then, steady-state availability

$$A = \lim_{t \to \infty} A(t) = \frac{\mu}{\lambda + \mu}$$

Fault-Tolerance Measures

MTBF

Mean time between failures

$$MTBF = \int_0^\infty R(t)\,dt$$

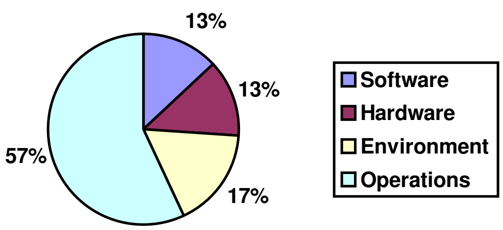
MTTR

Mean time to repair

Availability

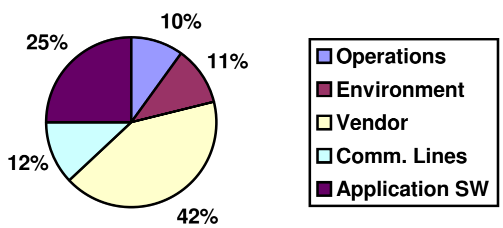
$$\frac{MTBF}{MTBF + MTTR}$$

Sources of Failure – SLAC Data (1985)



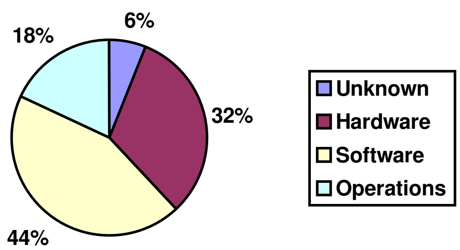
S. Mourad and D. Andrews, "The Reliability of the IBM/XA Operating System", *Proc. 15th Annual Int. Symp. On FTCS*, 1985.

Sources of Failure – Japanese Data (1986)



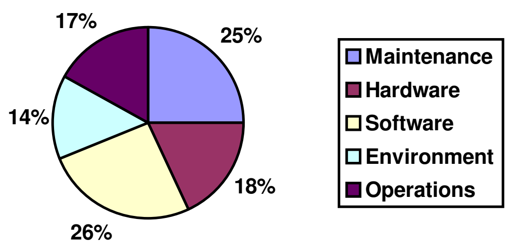
"Survey on Computer Security", Japan Info. Dev. Corp., 1986.

Sources of Failure – 5ESS Switch (1987)



D.A. Yaeger. 5ESS Switch Performance Metrics. *Proc. Int. Conf. On Communications, Volume 1*, pp. 46-52, June 1987.

Sources of Failures – Tandem Data (1985)



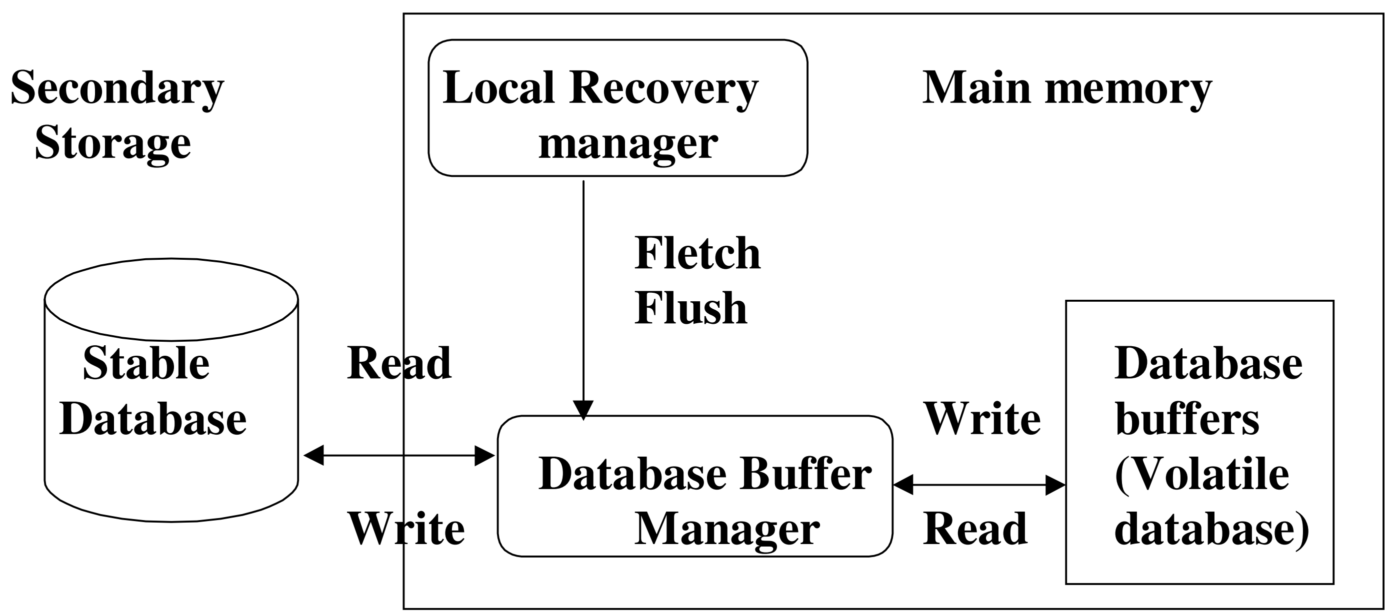
Jim Gray, *Why Do Computers Stop and What can be Done About it?*, Tandem Technical Report 85.7, 1985.

Types of Failures in D-DBMS

- Transaction failures
  - Transaction aborts (unilaterally or due to deadlock)
  - Avg. 3% of transactions abort abnormally
- System (site) failures
  - Failure of processors, main memory, power supply, …
  - Main memory contents are lost, but secondary storage contents are safe
  - Partial vs. total failure
- Media failures
  - Failure of secondary storage devices such that the stored data is lost
  - Head crash/controller failure(?)
- Communication failures
  - Lost/undeliverable messages
  - Network partitioning

Local Recovery Management – Architecture

- Volatile storage
  - ➢ Consists of the main memory of the computer system (RAM)
- Stable storage
  - ➢ Resilient to failures and loses its contents only in the presence of media failures (e.g., head crashes on disks).
  - ➢ Implemented via a combination of hardware (non-volatile storage) and software (stable-write, stable-read, clean-up) components



Update Strategies

- In-place update

  - Each update causes a change in one or more data values on pages in the database buffers

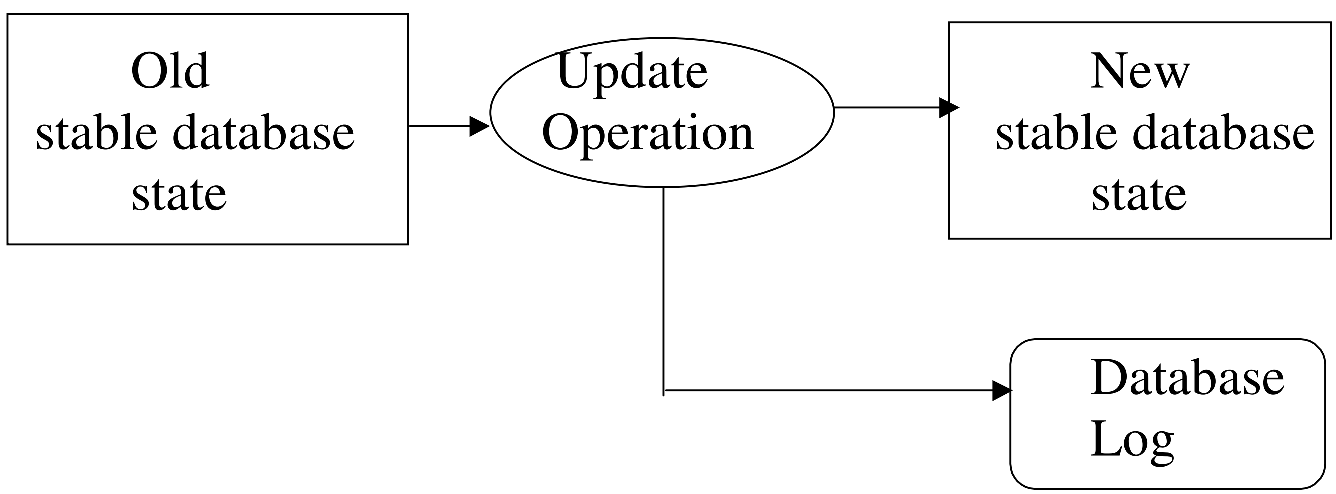
- Out-of-place update

  - Each update causes the new value(s) of data item(s) to be stored separate from the old value(s)

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

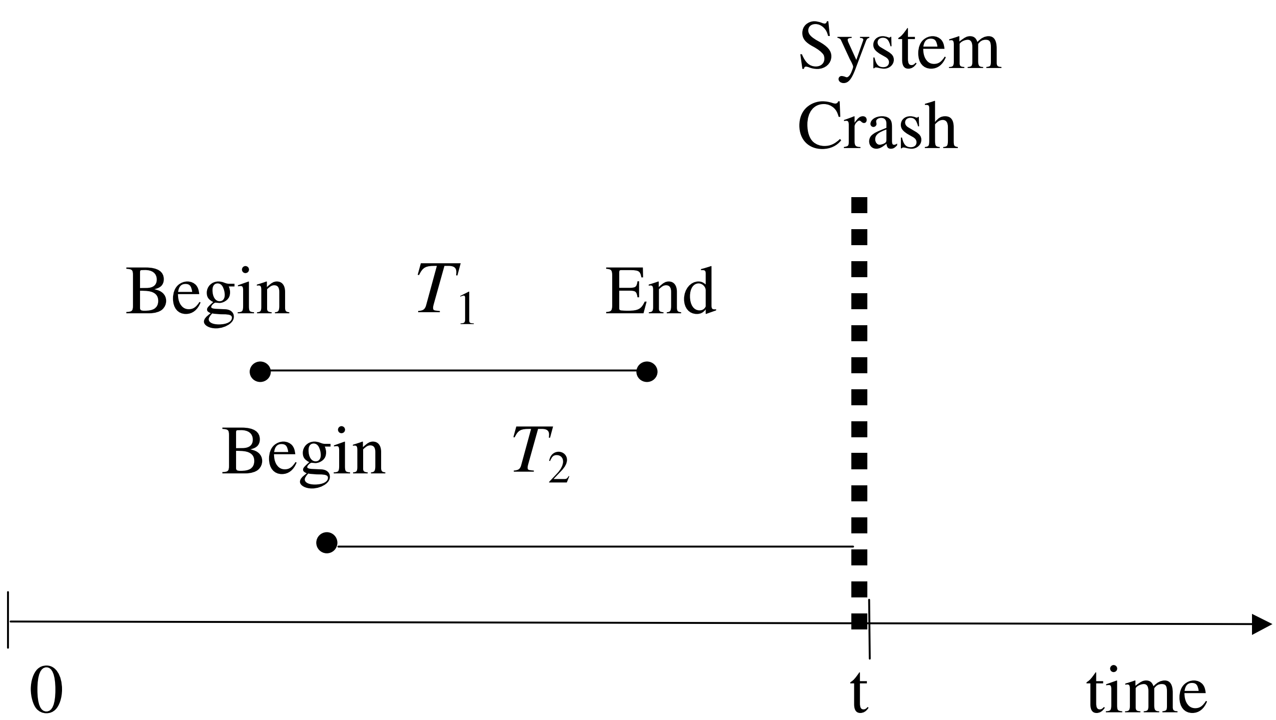
- transaction of identifier
- type of operation (action)
- items accessed by the transaction to perform the action
- old value (state) of item (before image)
- new value (state) of item (after image)

…

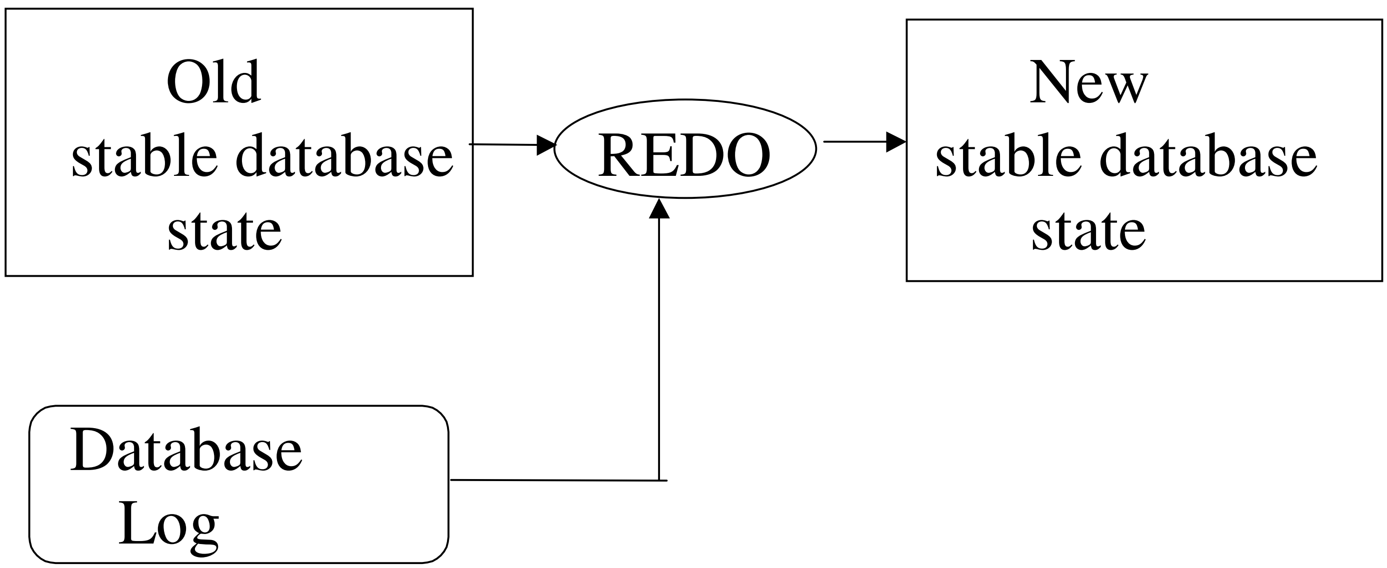
Why Logging?

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)

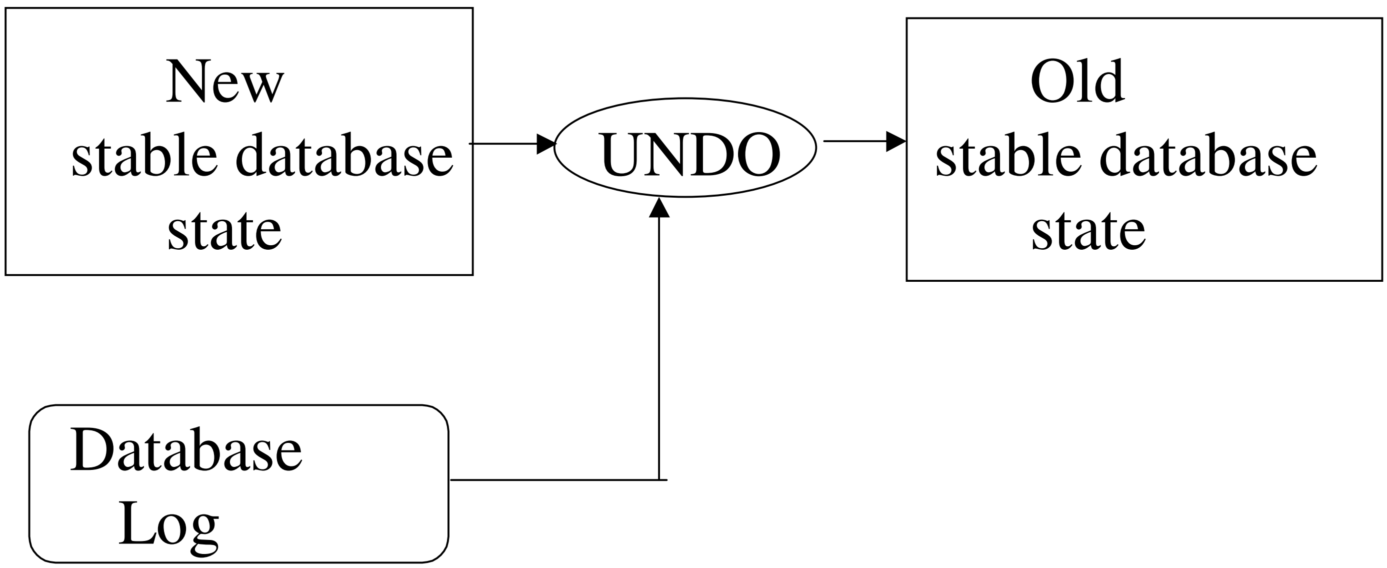


REDO Protocol



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

UNDO Protocol



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

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)

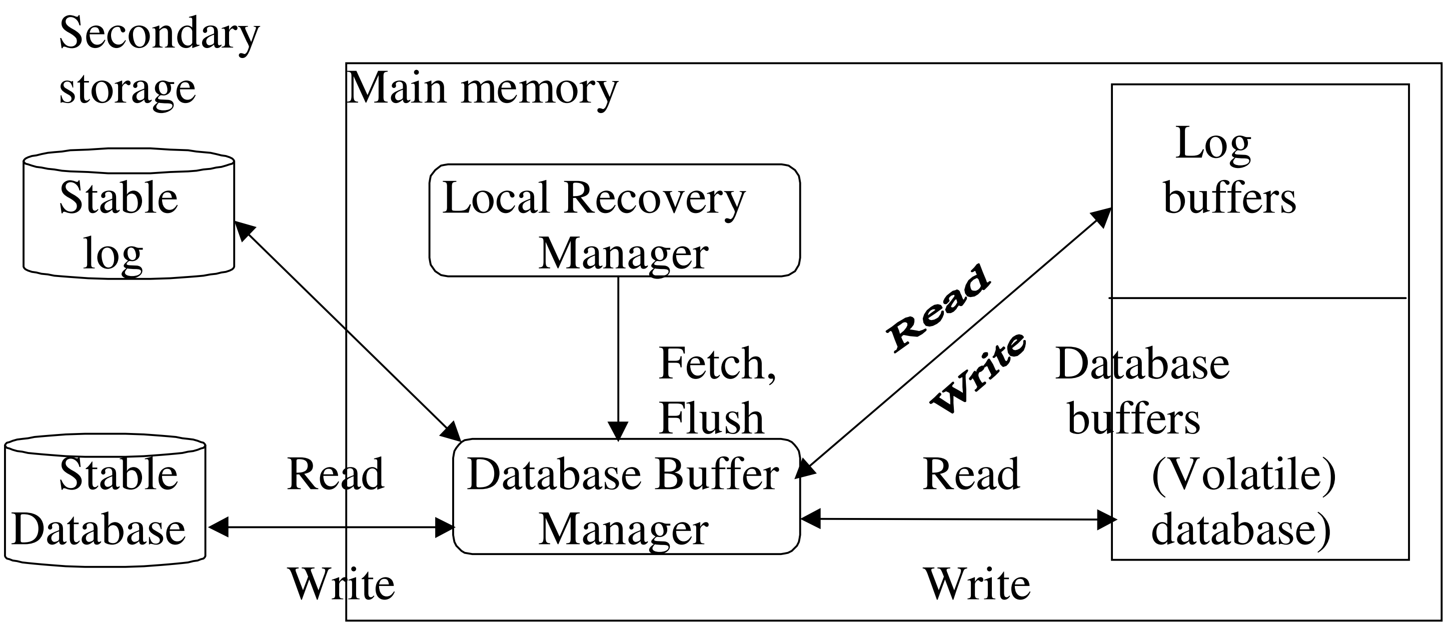
  We can recover (undo) by restoring *P* to its old state by using the log
- Unfortunate case
  - System writes *P* in stable database
  - SYSTEM FAILURES OCCURS!… (before stable log is updated)

  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 committee, some of its actions might have to be redone. Need the after images (*redo portion* of the 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.

Logging Interface



Out-of-Place Update Recovery Information

- 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 retained for recovery.

- Differential files
  - For each file F maintain
    - a read only FR
    - a differential file consisting of insertions part DF+ and delections part DF-
    - Thus, F = (FR ∪ DF+) – DF-
  - Updates treated as delete old value, insert new value
  -

FR: the stable database part. F: The logial database file.

Execution of Commands

Commands to consider:

begin_transaction
read
write
commit
abort
recover

Independent of execution strategy for LRM

LRM: Local recovery Manager

Execution Strategies

- Dependent upon
  - Can the buffer manager 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
  - 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
- Possible execution strategies:
  - no-fix/no-flush (undo/redo)
  - no-fix/flush (undo/no redo)
  - fix/no-flush (no undo/redo)
  - fix/flush (no undo/no redo)

No-Fix/No-Flush (undo/redo)

- Abort
  - Buffer manager may have written some of the updated pages into stable database
  - LRM performs transaction undo (or partial undo)
- Commit
  - LRM writes an “end_of_transaction” record into the log.
- Recover
  - For those transactions that have both a “begin_transaction” and an “end_of_transaction” record in the log, a partial redo is initiated by LRM
  - For those transactions that only have a “begin_transaction” in the log, a global undo is executed by LRM

No-Fix/Flush (undo/no redo)

- Abort
  - Buffer manager may have written some of the updated pages into stable database
  - LRM performs transaction undo (or partial undo)
- Commit
  - LRM issues a **flush** command to the buffer manager for all updated pages
  - LRM writes an "end_of_transaction" record into the log.
- Recover
  - No need to perform redo
  - Perform global undo

Fix/No-Flush **(no undo/redo)**

- Abort
  - None of the updated pages have been written into stable database
  - Release the **fix**ed pages
- Commit
  - LRM writes an “end_of_transaction” record into the log.
  - LRM sends an **unfix** command to the buffer manager for all pages that were previously **fix**ed
- Recover
  - Perform partial redo
  - No need to perform global undo

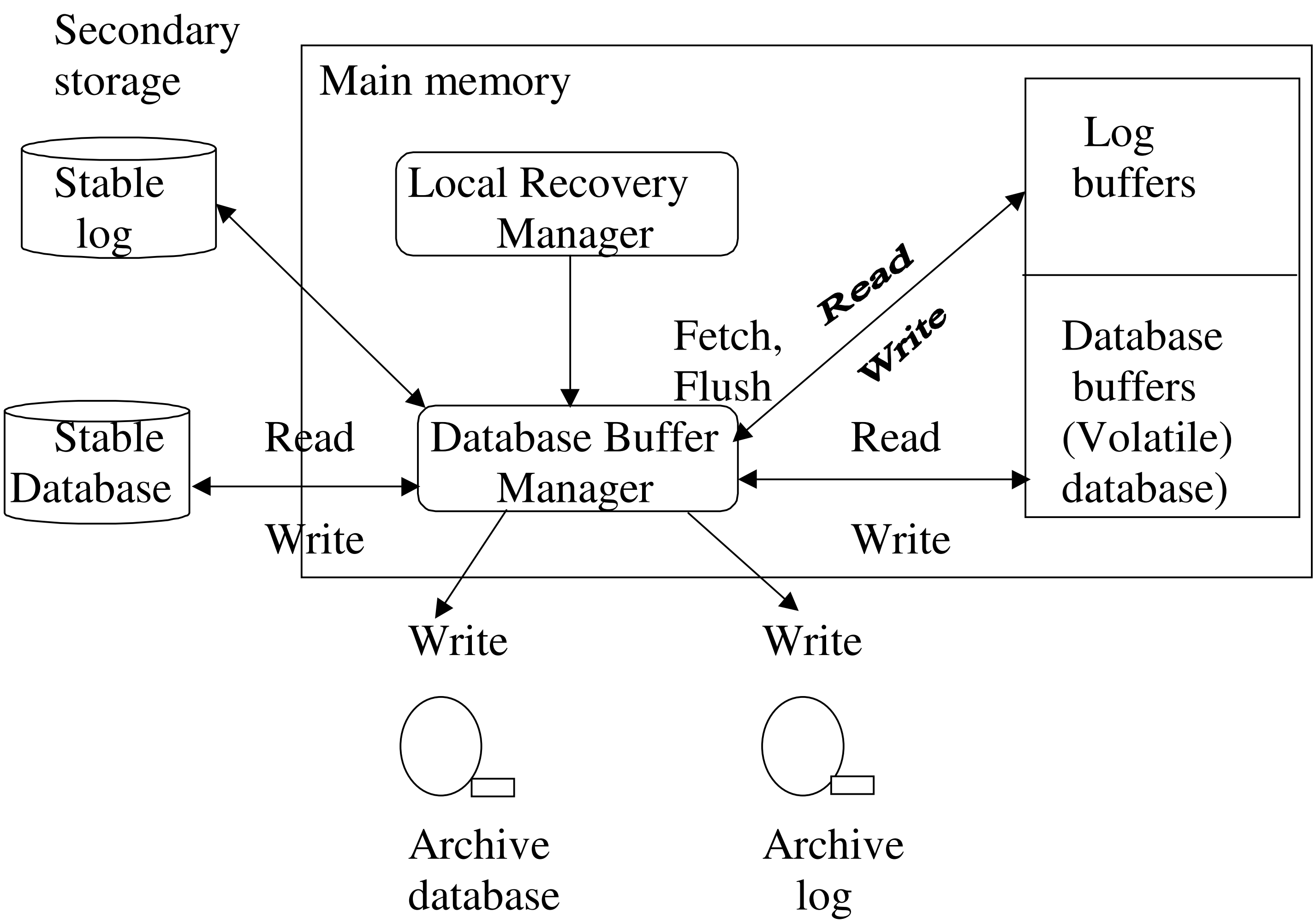
Fix/Flush (no undo/no redo)

- Abort
  - None of the updated pages have been written into stable database
  - Release the **fix**ed pages
- Commit (the following have to be done atomically)
  - LRM issues a **flush** command to the buffer manager for all updated pages
  - LRM sends an **unfix** command to the buffer manager for all pages that were previously **fix**ed
  - 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.
- A checkpoint record contains a list of active transactions.
- Steps:
  - Write a begin_checkpoint record into the log
  - Collect the checkpoint data into the stable storage
  - 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 can the remaining 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.
- Independent recovery $\Rightarrow$ non-blocking termination

Two-Phase Commit (2PC)

Phase 1: The coordinator gets the participants ready to write the results into the database

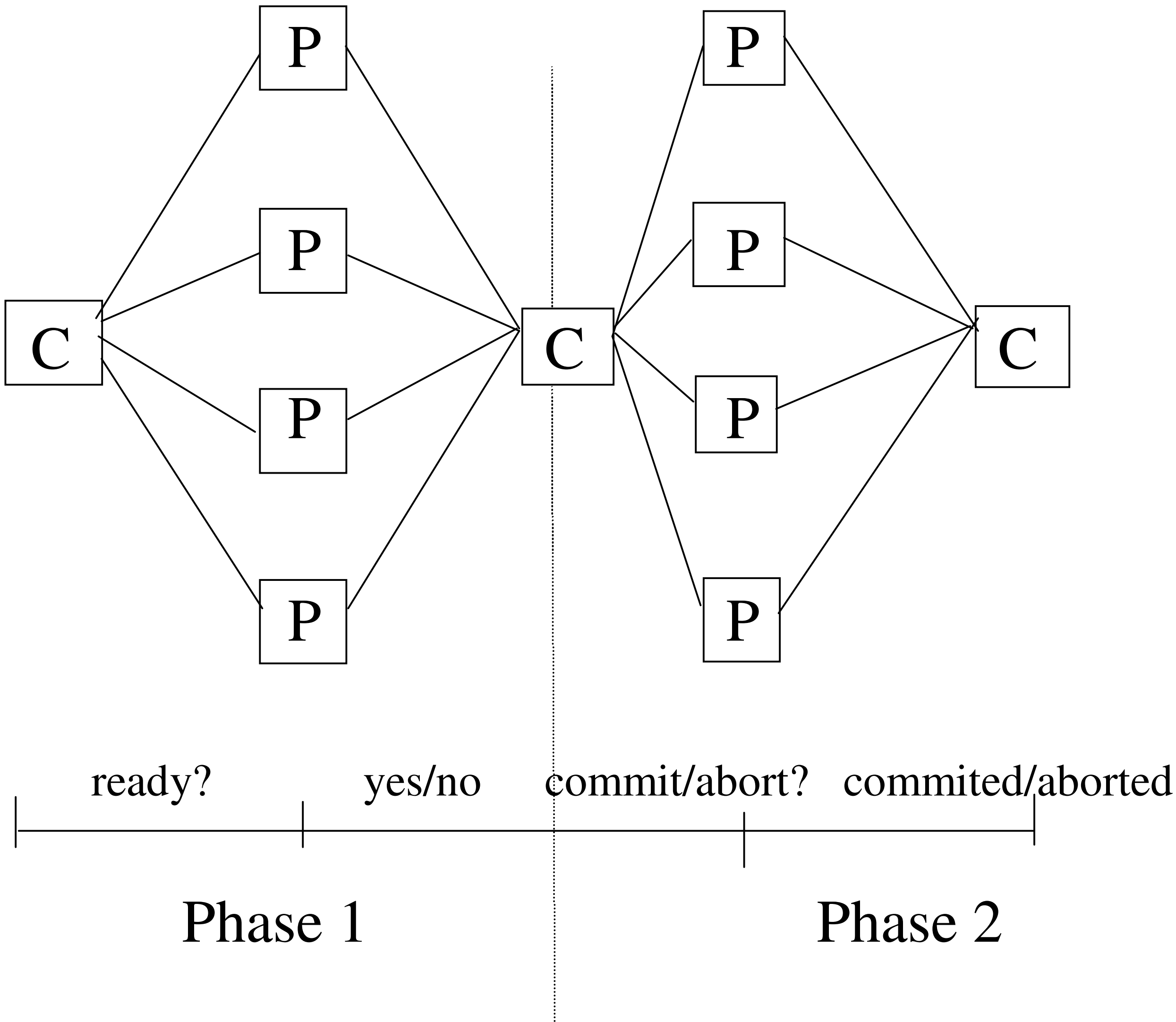
Phase 2: Everybody writes the results into the database

- **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

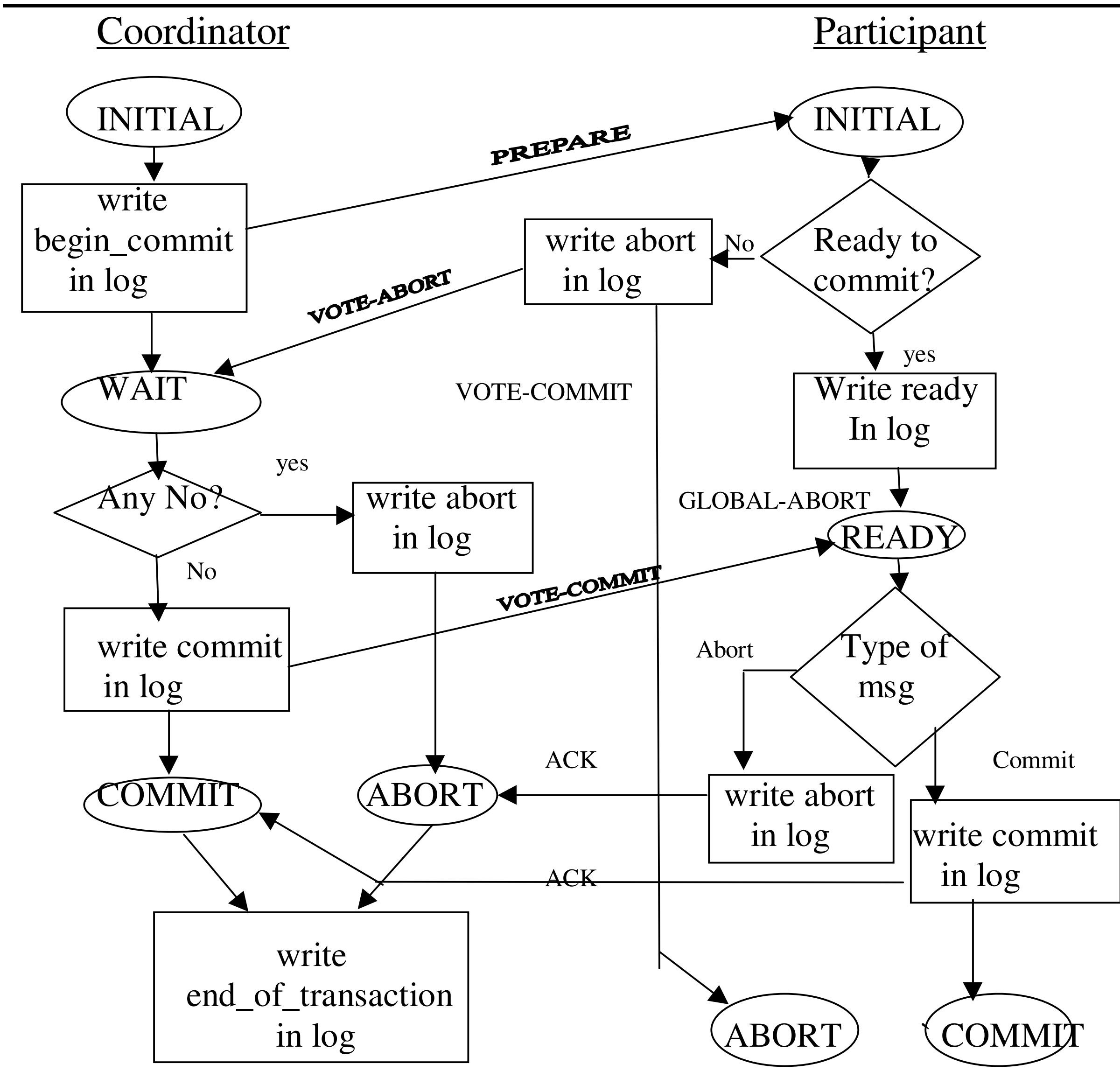
Global Commit Rule:

- 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 2PC

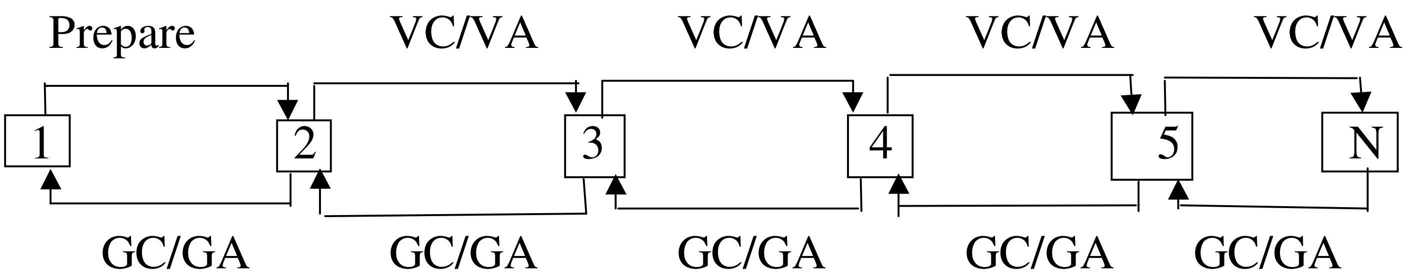


2PC Protocol Actions



Linear 2PC

Phase 1

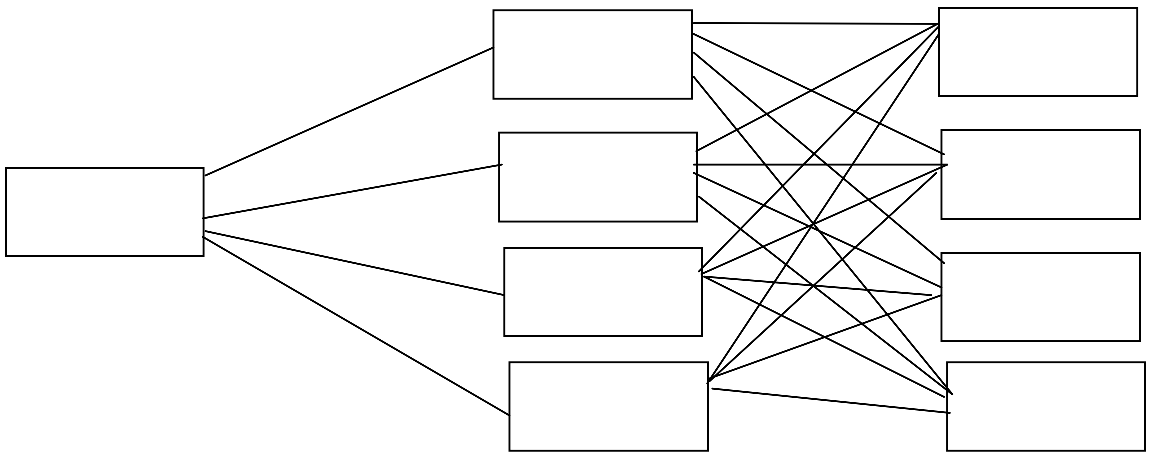


Phase 2

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

Distributed 2PC

Coordinator   Participants   Participants



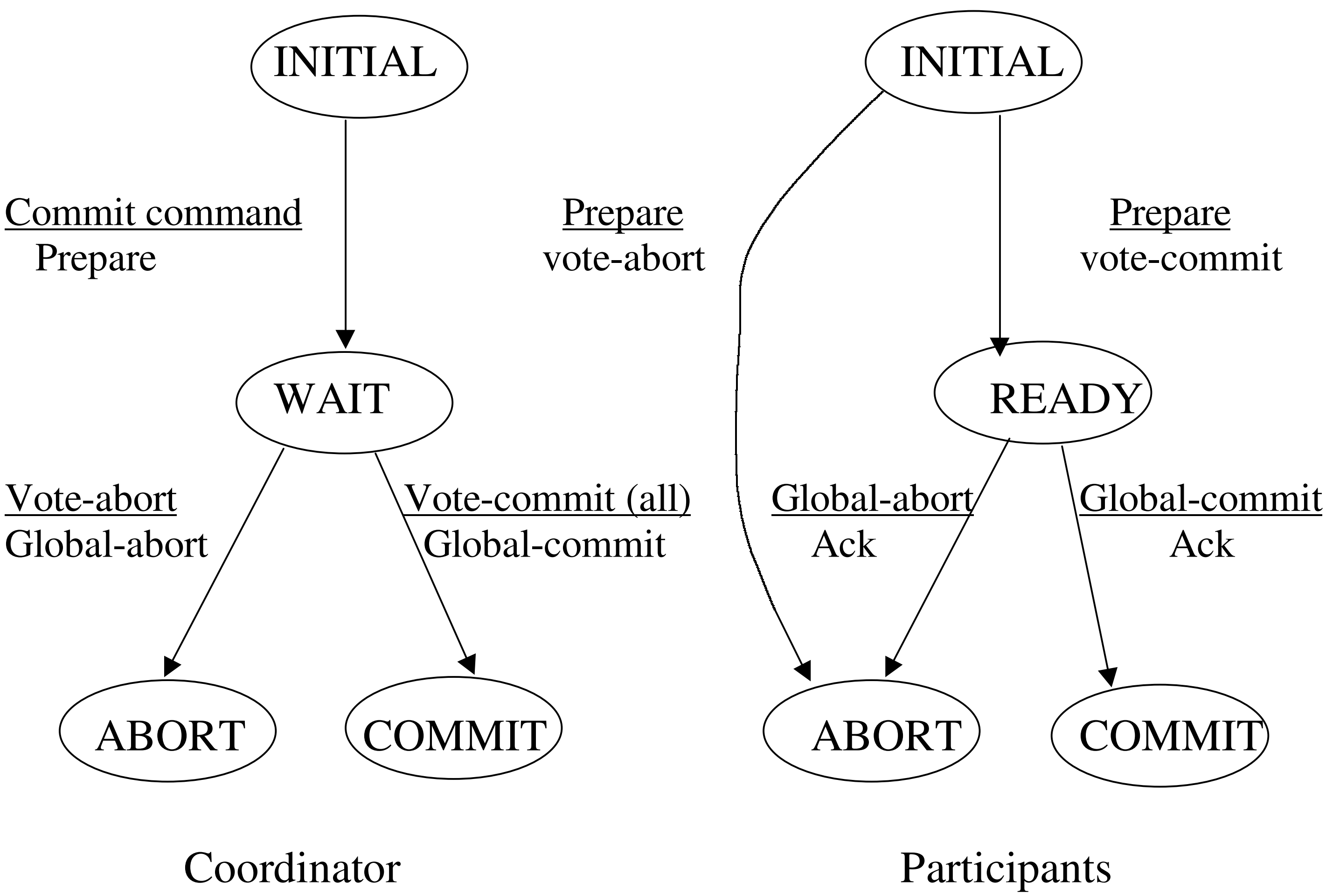
Prepare

vote-abort/
vote-commit

global-commit/
global-abort
decision made
independently

Phase 1

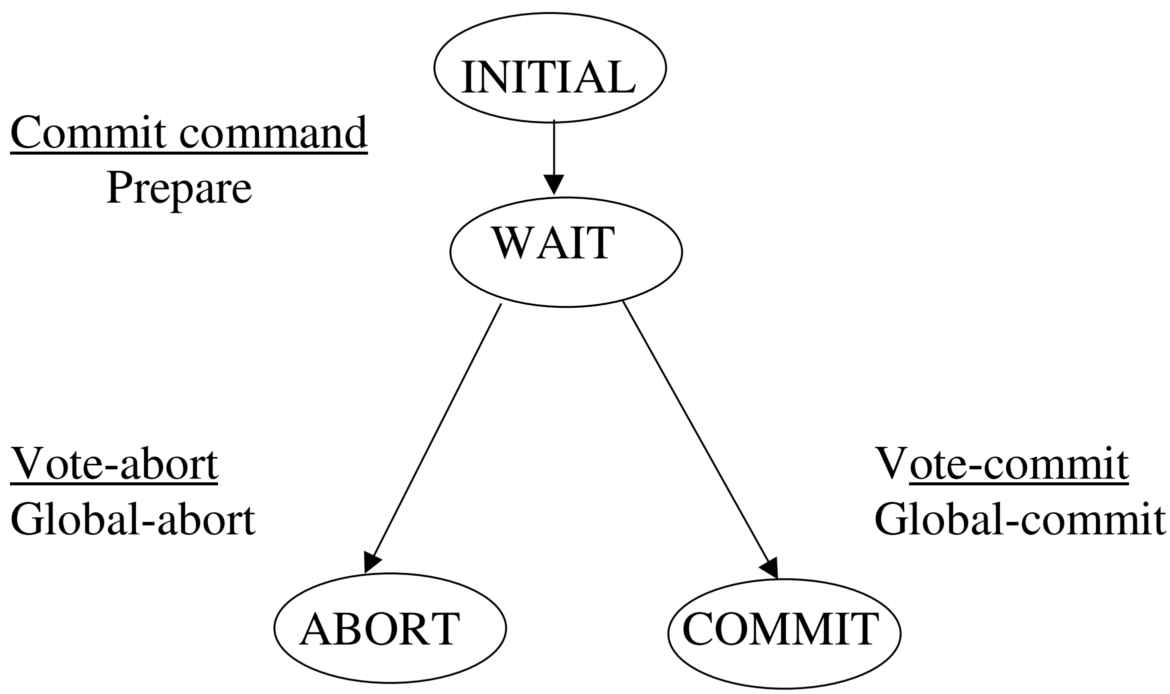
State Transitions in 2PC



Site Failures – 2PC Termination

- Timeout in INITIAL
  - Who cares
- Timeout in WAIT
  - Cannot unilaterally commit
  - Can unilaterally abort
- Timeout in ABORT or COMMIT
  - Stay blocked and wait for the acks

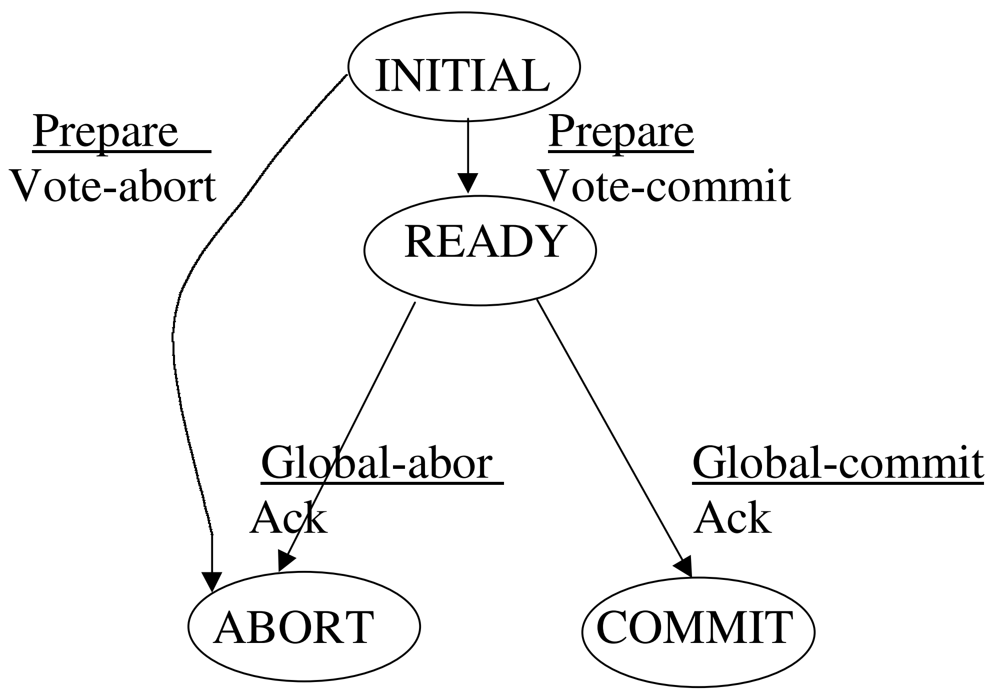
COORDINATOR



Site Failures – 2PC Termination

- Timeout in INITIAL
  - Coordinator must have failed in INITIAL state
  - Unilaterally abort
- Timeout in READY
  - Stay blocked

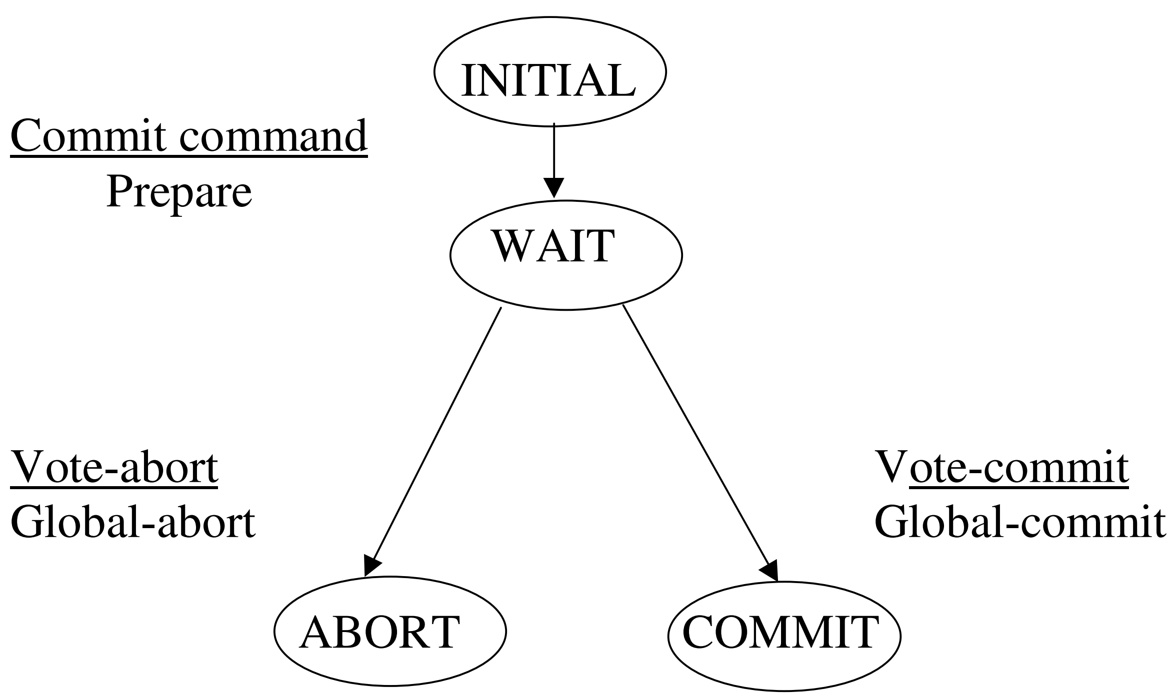
PARTICIPANTS



Site Failures – 2PC Recovery

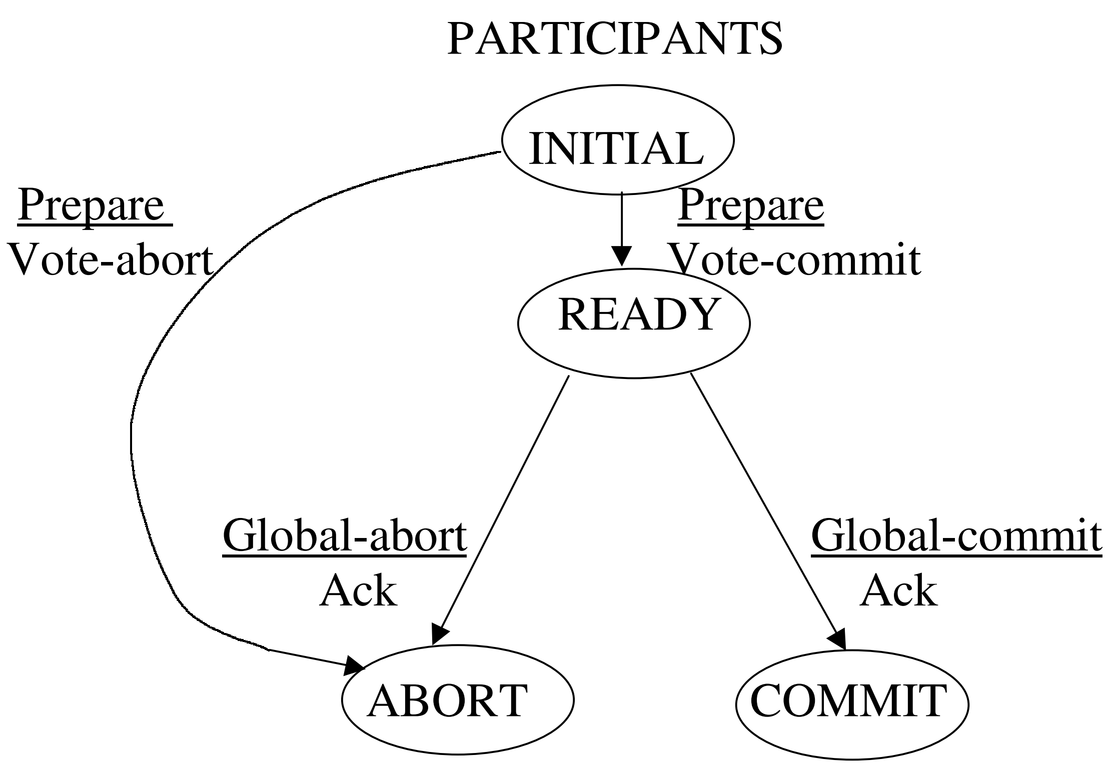
- Failure in INITIAL
  - ➢ Start the commit process upon recovery
- Failure in WAIT
  - ➢ Restart the commit process upon recovery
- Failure in ABORT or COMMIT
  - ➢ Nothing special if all the acks have been received
  - ➢ Otherwise the termination protocol is involved

COORDINATOR



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



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 send "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

2PC Recovery Protocols – Additional Case

- 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

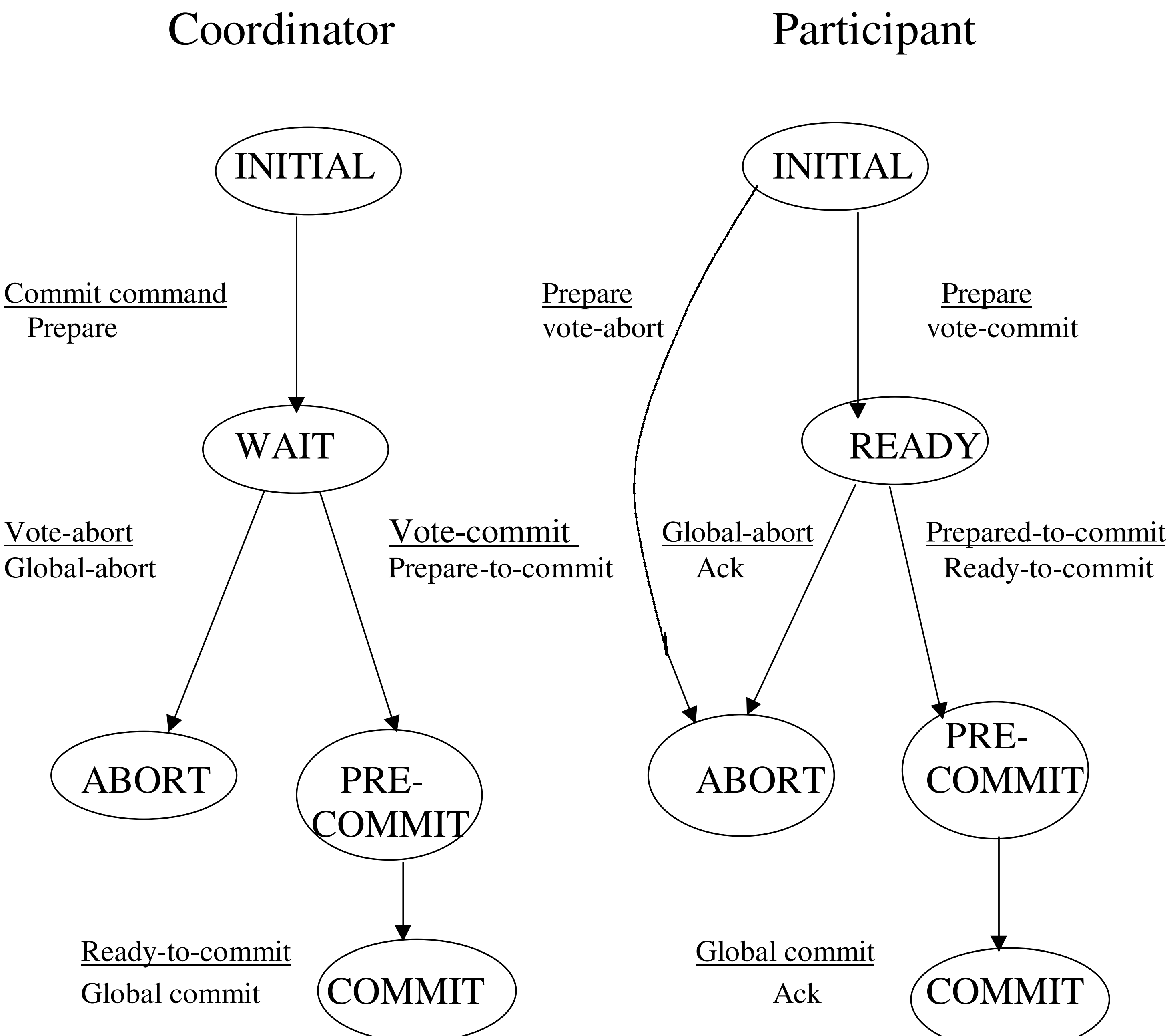
Problem With 2PC

- Blocking
  - Ready implies that the participant waits for the coordinator
  - If coordinator fails, site is blocked until recovery
  - Blocking reduces availaibilty
- 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
- So we search for these protocol – 3PC

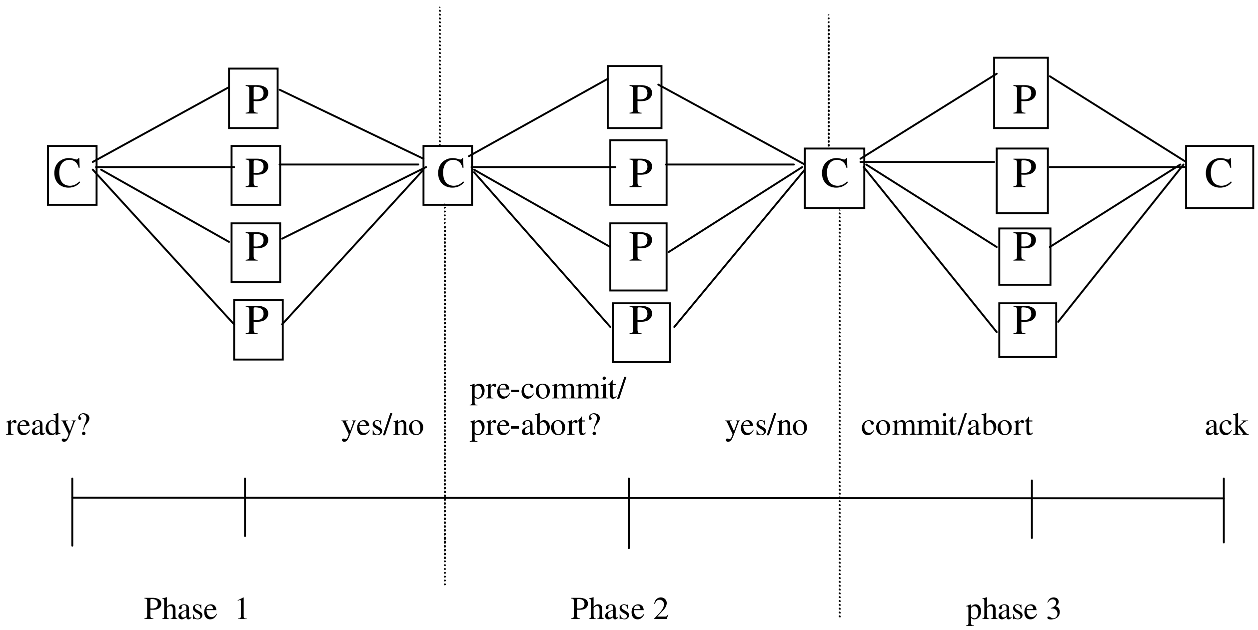
Three-Phase Commit

- 3PC is non-blocking
- A commit protocols is non-blocking iff
  - ➢ it is synchronous within one state transition, and
  - ➢ its state transition diagram contains
    - ♦ no state which is “adjacent” to both a commit and an abort state, and
    - ♦ no non-committable state which is “adjacent” to a commit state
- Adjacent: possible to go from one state to another with a single state transition
- Committable: all sites have voted to commit a transaction
  - ➢ e.g.: COMMIT state

State Transitions in 3PC

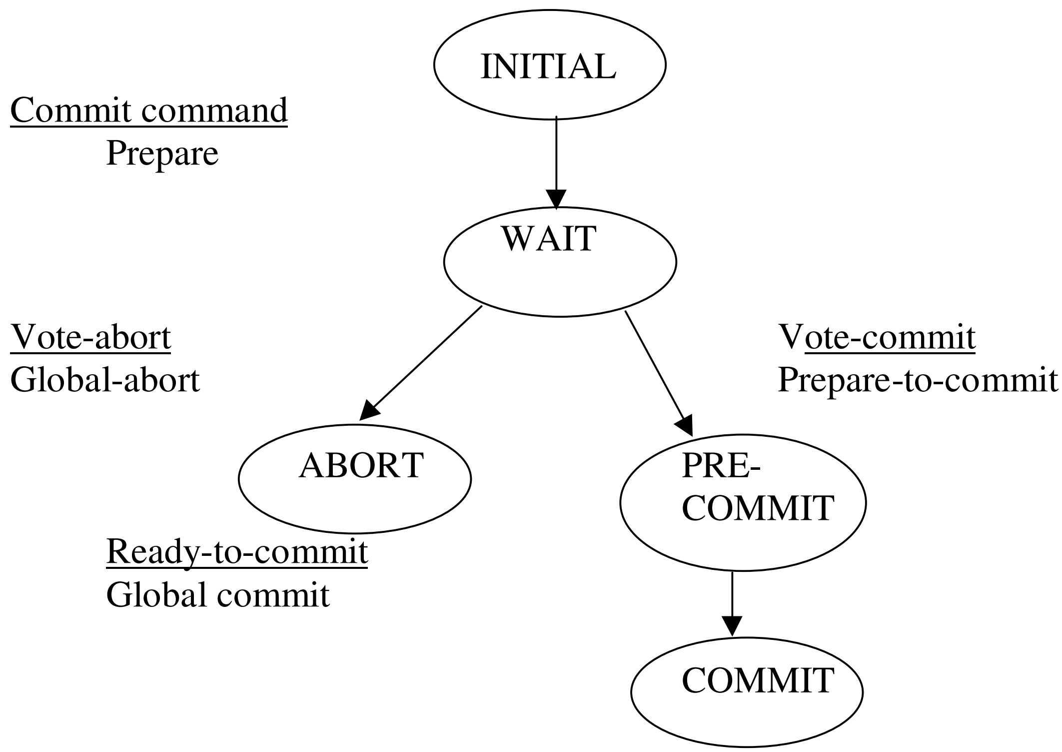


Communication Structure



Site Failures – 3PC Termination

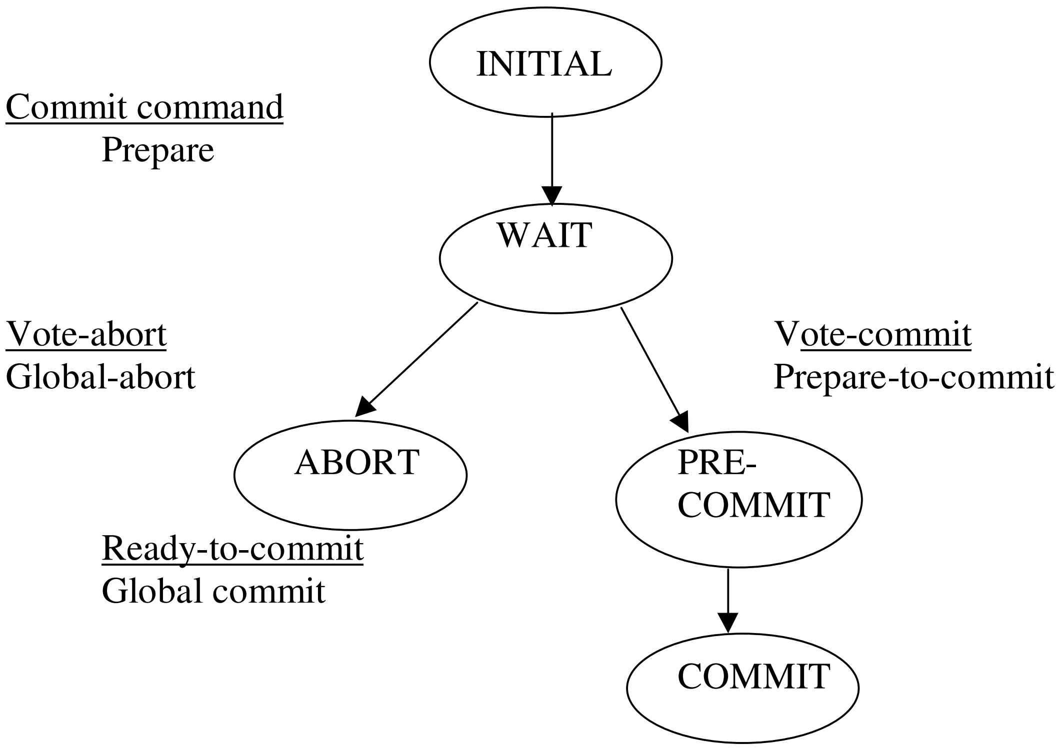
COORDINATOR



- Timeout in INITIAL
  - Who cares
- Timeout in WAIT
  - Unilaterally abort
- Timeout in PRECOMMIT
  - Participants may not be in PRECOMMIT, but at least in READY
  - Move all the participants to PRECOMMIT state
  - Terminate by globally committing

Site Failures – 3PC Termination

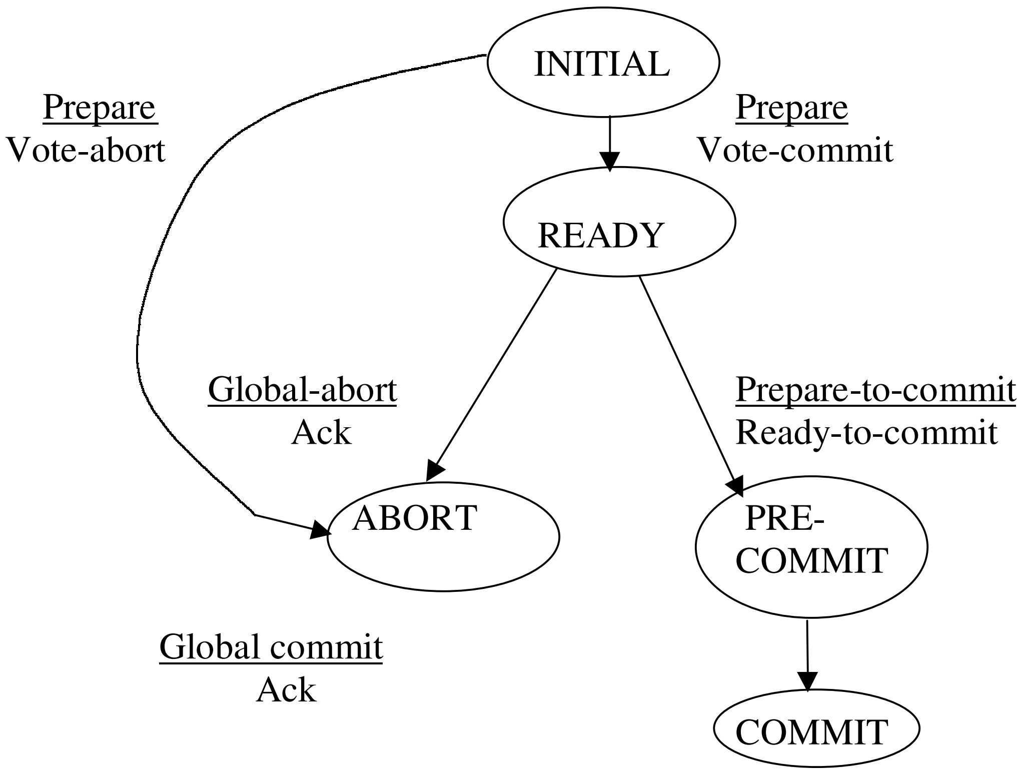
COORDINATOR



- Timeout in ABORT or COMMIT
  - Just ignore and treat the transaction as completed
  - participants are either in PRECOMMIT or READY state and can follow their termination protocols

Site Failures – 3PC Termination

PARTICIPANTS



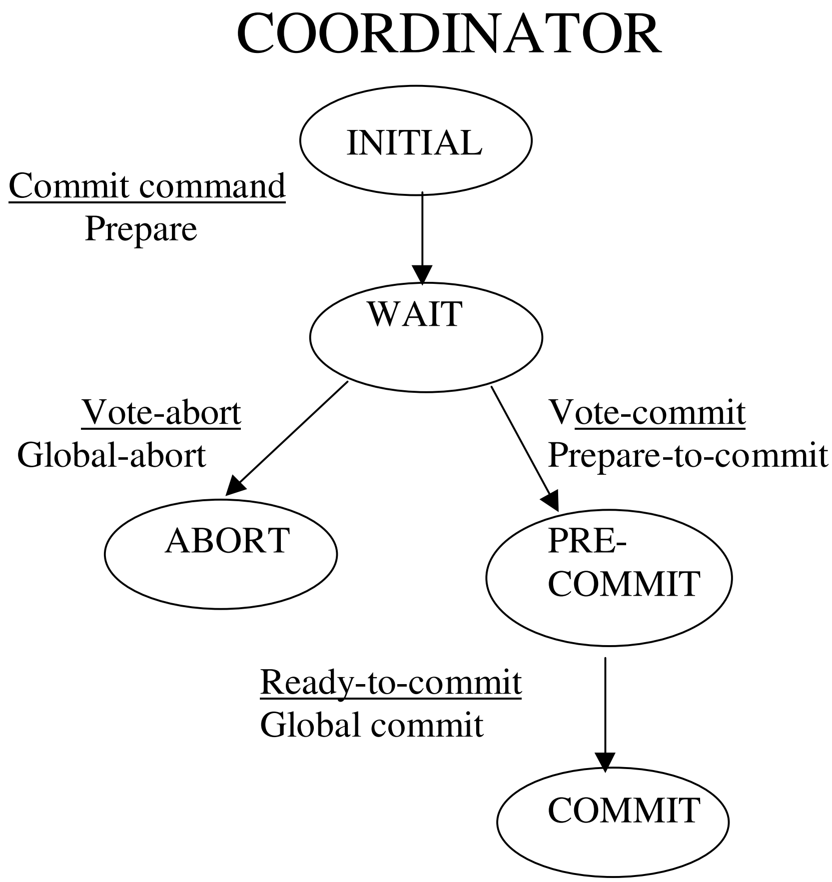
- Timeout in INITIAL
  - Coordinator must have failed in INITIAL state
  - Unilaterally abort
- Timeout in READY
  - Voted to commit, but does not know the coordinator's decision
  - Elect a new coordinator and terminate using a special protocol
- Timeout in PRECOMMIT
  - Handle it the same as timeout in READY state

Termination Protocol Upon Coordinator Election

New coordinator can be in one of four states: WAIT, PRECOMMIT, COMMIT, ABORT

- Coordinator sends its state to all of the participants asking them to assume its state.
- Participants "back-up" and reply with appriate messages, except those in ABORT and COMMIT states. Those in these states respond with "Ack" but stay in their states.
- Coordinator guides the participants towards termination:
  - ♦ If the new coordinator is in the WAIT state, participants can be in INITIAL, READY, ABORT or PRECOMMIT states. New coordinator globally aborts the transaction.
  - ♦ If the new coordinator is in the PRECOMMIT state, the participants can be in READY, PRECOMMIT or COMMIT states. The new coordinator will globally commit the transaction.
  - ♦ If the new coordinator is in the ABORT or COMMIT states, at the end of the first phase, the participants will have moved to that state as well.

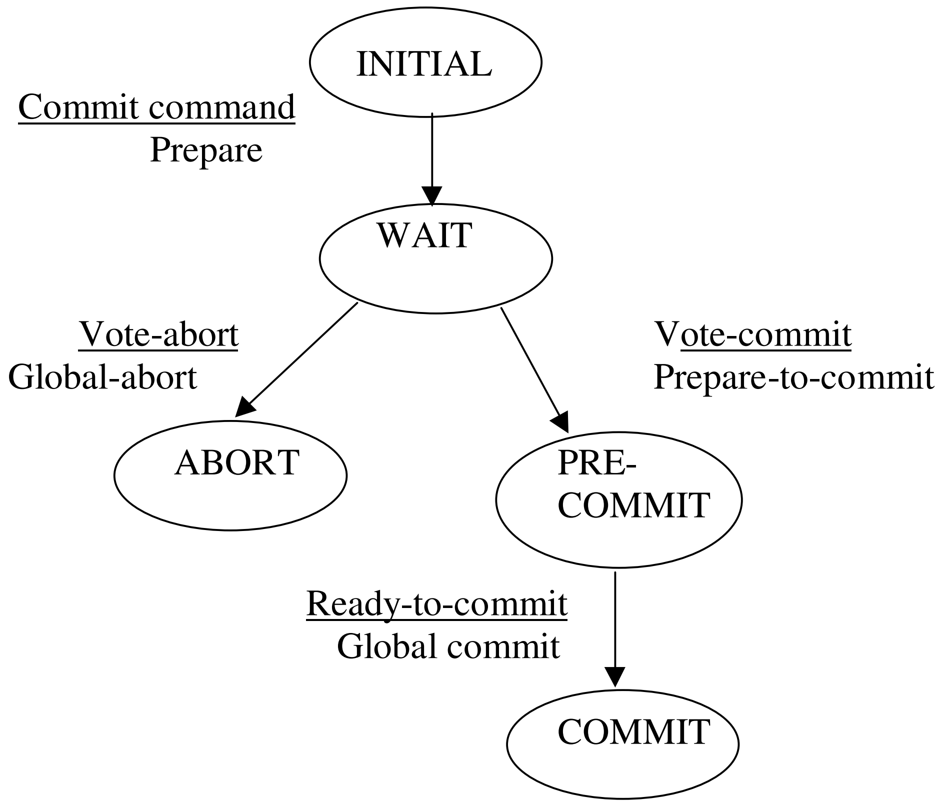
Site Failures – 3PC Recovery



- Failure in INITIAL
  - Start commit process upon recovery
- Failure in WAIT
  - the participants may have elected a new coordinator and terminated the transaction
  - the new coordinator could be in WAIT or ABORT states ⇒ transaction aborted
  - ask around for the fate of the transaction
- failure in PRECOMMIT
  - ask around for the fate of the transaction

Site Failures – 3PC Recovery

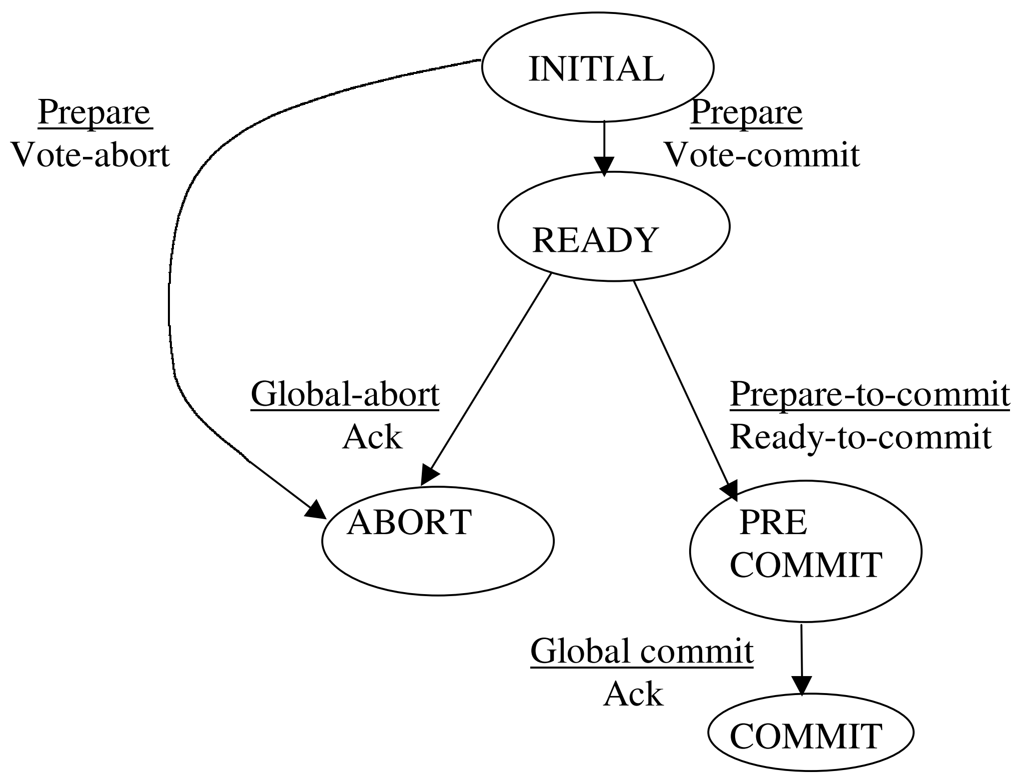
COORDINATOR



- Failure in COMMIT or ABORT
  - ➢ Nothing special if all the acknowledgements have been received; otherwise the termination protocol is involved

Site Failures – 3PC Recovery

PARTICIPANTS



- Failure in INITIAL
  - unilaterally abort upon recovery
- Failure in READY
  - the coordinator has been informed about the local decision
  - upon recovery, ask around
- Failure in PRECOMMIT
  - ask around to determine how the other participants have terminated the transaction
- Failure in COMMIT or ABORT
  - no need to do anything

Network Partitioning

- Simple partitioning
  - Only two partitions

- Multiple partitioning
  - More than two partitions

- Formal bounds (due to Skeen):
  - There exists no non-blocking protocols that is resilient to a network partition if messages are lost when partition occurs.
  - There exist non-blocking protocols which are resilient to a single network partition if all undeliverable messages are returned to sender.
  - There exists no non-blocking protocol which is resilient to a multiple partition.

Independent Recovery Protocols for Network Partitioning

- No general solution possible
  - allow one group to terminate while the other is blocked
  - improve availability

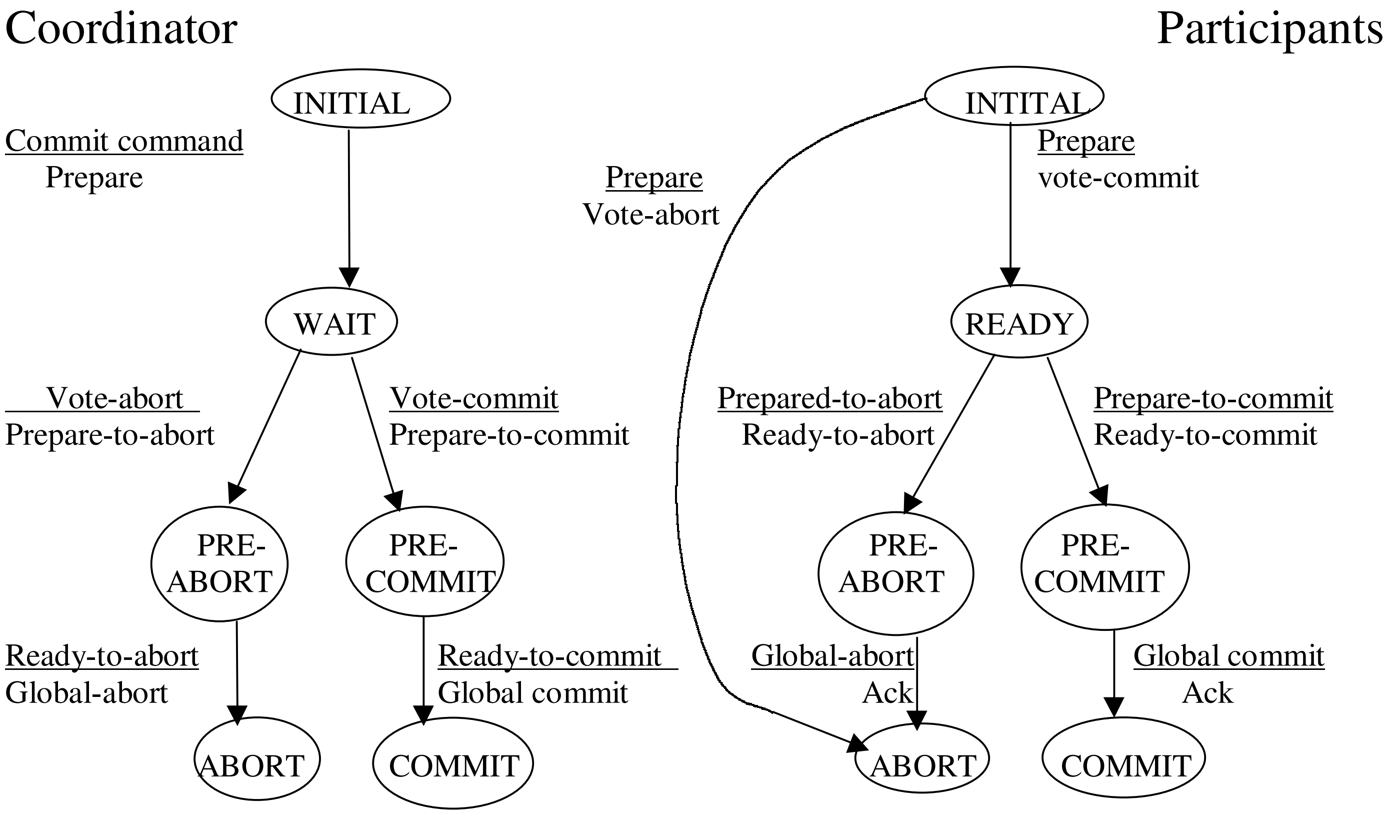
- How to determine which group to proceed?
  - The group with a majority

- How does a group know if it has majority?
  - centralized
    - whichever partitions contains the central site should terminate the transaction
  - voting-based (quorum)
    - different for replicated vs non-replicated databases

Quorum Protocols for Non-Replicated Databases

- The network partitioning problem is handled by the commit protocol.
- Every site is assigned a vote V_i
- Total number of votes in the system V
- `Abort quorum V_a, commit quorum V_c
  - $V_a + V_c > V$ where $0 \leq V_a, V_c \leq V$
  - Before a transaction commits, it must obtain a commit quorum V_c
  - Before a transaction aborts, it must obtain an abort quorum V_a

State Transitions in Quorum Protocols



Quorum Protocols for Replicated Databases

- Network partitioning is handled by the replica control protocol.
- One implementation:
  - ➢ Assign a vote to each *copy* of a replicated data item (say V_i) such that $\Sigma_i \mathrm{V}_i = V$
  - ➢ Each operation has to obtain a *read quorum* (V_r) to read and a *write quorum* (V_w) to write a data item
  - ➢ Then the following rules have to be obeyed in determining the quorums:
    - ♦ $V_r + V_w > V$ a data item is not read and written by two transactions concurrently
    - ♦ $V_w > V/2$ two write operations from two transactions cannot occur concurrently on the same data item

Use for Network Partitioning

- Simple modification of the ROWA rule:
  - When the replica control protocol attempts to read or write a data item, it first checks if a majority of the sites are in the same partition as the site that the protocol is running on (by checking its votes). If so, execute the ROWA rule within that partition.
- Assumes that failures are "clean' which means:
  - failures that change the network's topology are detected by all sites instantaneously
  - each site has a view of the network consisting of all the sites it can communicate with

Open problems

- Replication protocols
  - experimental validation
  - replication of computation and communication
- Transaction models
  - changing requirements
    - cooperative sharing vs. competitive sharing
    - interactive transactions
    - longer duration
    - complex operations on complex data
  - relaxed semantics
    - non-serializable correctness criteria

Transaction Model Design Space