Unit3
Fault Tolerance

System reliability: Fault-Intolerance vs. Fault-Tolerance

- The fault intolerance (or fault-avoidance) approach improves system reliability by removing the source of failures (i.e., hardware and software faults) before normal operation begins

- The approach of fault-tolerance expect faults to be present during system operation, but employs design techniques which insure the continued correct execution of the computing process

Approaches to fault-tolerance

Approaches:

(a) Mask failures

(b) Well defined failure behavior

(a) Mask failures:

- System continues to provide its specified function(s) in the presence of failures
- Example: voting protocols

(b) Well defined failure behaviour:

- System exhibits a well define behaviour in the presence of failures
- It may or it may not perform its specified function(s), but facilitates actions suitable for fault recovery
- Example: commit protocols
  - A transaction made to a database is made visible only if successful and it commits
  - If it fails, transaction is undone

Redundancy:

- Method for achieving fault tolerance (multiple copies of hardware, processes, data, etc...)

Issues

- Process Deaths:
  - All resources allocated to a process must be recovered when a process dies
  - Kernel and remaining processes can notify other cooperating processes
  - Client-server systems: client (server) process needs to be informed that the corresponding server (client) process died
- Machine failure:
  - All processes running on that machine will die
  - Client-server systems: difficult to distinguish between a process and machine failure
  - Issue: detection by processes of other machines
- Network Failure:
  - Network may be partitioned into subnets
  - Machines from different subnets cannot communicate
  - Difficult for a process to distinguish between a machine and a communication link failure

Atomic actions

- System activity: sequence of primitive or atomic actions
- Atomic Action:
  - Machine Level: uninterruptible instruction
  - Process Level: Group of instructions that accomplish a task
  - Example: Two processes, P1 and P2, share a memory location ‘x’ and both modify ‘x’

```
Process P1   Process P2
...          ...
Lock(x);           Lock(x);
x := x + z;        x := x + y;
Unlock(x);   Unlock(x);
...          ...
             successful exit
```

Atomic action

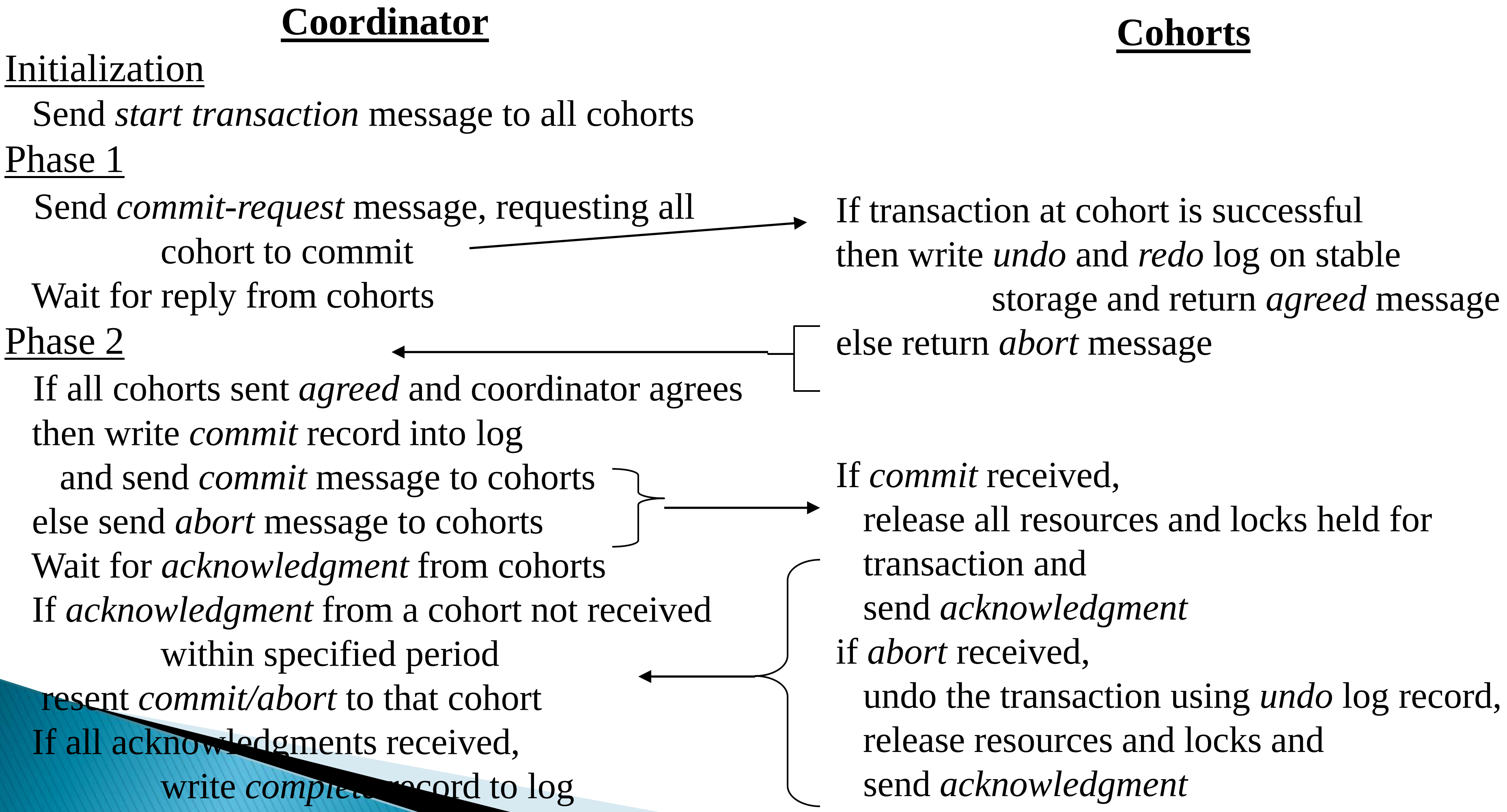
  - System level: group of cooperating process performing a task (global atomicity)

Committing

- Transaction: Sequence of actions treated as an atomic action to preserve consistency (e.g. access to a database)
- Commit a transaction: Unconditional guarantee that the transaction will complete successfully (even in the presence of failures)
- Abort a transaction: Unconditional guarantee to back out of a transaction, i.e., that all the effects of the transaction have been removed (transaction was backed out)
  - Events that may cause aborting a transaction: deadlocks, timeouts, protection violation
  - Mechanisms that facilitate backing out of an aborting transaction
    - Write-ahead-log protocol
    - Shadow pages
- Commit protocols:
  - Enforce global atomicity (involving several cooperating distributed processes)
  - Ensure that all the sites either commit or abort transaction unanimously, even in the presence of multiple and repetitive failures

The two-phase commit protocol

- Assumption:
  - One process is coordinator, the others are “cohorts” (different sites)
  - Stable store available at each site
  - Write-ahead log protocol



Coordinator

Initialization

Send *start transaction* message to all cohorts

Phase 1

Send *commit-request* message, requesting all cohort to commit

Wait for reply from cohorts

Phase 2

If all cohorts sent *agreed* and coordinator agrees

then write *commit* record into log

and send *commit* message to cohorts

else send *abort* message to cohorts

Wait for *acknowledgment* from cohorts

If *acknowledgment* from a cohort not received within specified period

resent *commit/abort* to that cohort

If all acknowledgments received,

write *complete* record to log

Cohorts

If transaction at cohort is successful

then write *undo* and *redo* log on stable storage and return *agreed* message

else return *abort* message

If *commit* received,

release all resources and locks held for transaction and

send *acknowledgment*

if *abort* received,

undo the transaction using *undo* log record,

release resources and locks and

send *acknowledgment*

Voting protocols

- Principles:
  - Data replicated at several sites to increase reliability
  - Each replica assigned a number of votes
  - To access a replica, a process must collect a majority of votes
- Vote mechanism:

  (1) Static voting:

  - Each replica has number of votes (in stable storage)
  - A process can access a replica for a read or write operation if it can collect a certain number of votes (*read* or *write quorum*)

  (2) Dynamic voting

  - Number of votes or the set of sites that form a quorum change with the state of system (due to site and communication failures)

  (2.1) Majority based approach:

    - Set of sites that can form a majority to allow access to replicated data of changes with the changing state of the system

  (2.2) Dynamic vote reassignment:

    - Number of votes assigned to a site changes dynamically

Failure resilient processes

- Resilient process: continues execution in the presence of failures with minimum disruption to the service provided (masks failures)
- Approaches for implementing resilient processes:
  - Backup processes and
  - Replicated execution

(1) Backup processes

- Each process made of a primary process and one or more backup processes
- Primary process execute, while the backup processes are inactive
- If primary process fails, a backup process takes over
- Primary process establishes checkpoints, such that backup process can restart

(2) Replicated execution

- Several processes execute same program concurrently
- Majority consensus (voting) of their results
- Increases both the reliability and availability of the process

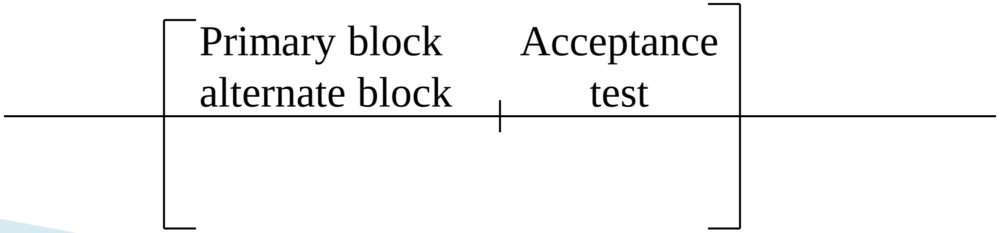
Recovery (fault tolerant) block concept

- Provide fault-tolerance within an individual sequential process in which assignments to stored variables are the only means of making recognizable progress
- The recovery block is made of:
  - A primary block (the conventional program),
  - Zero or more alternates (providing the same function as the primary block, but using different algorithm), and
  - An acceptance test (performed on exit from a primary or alternate block to validate its actions).

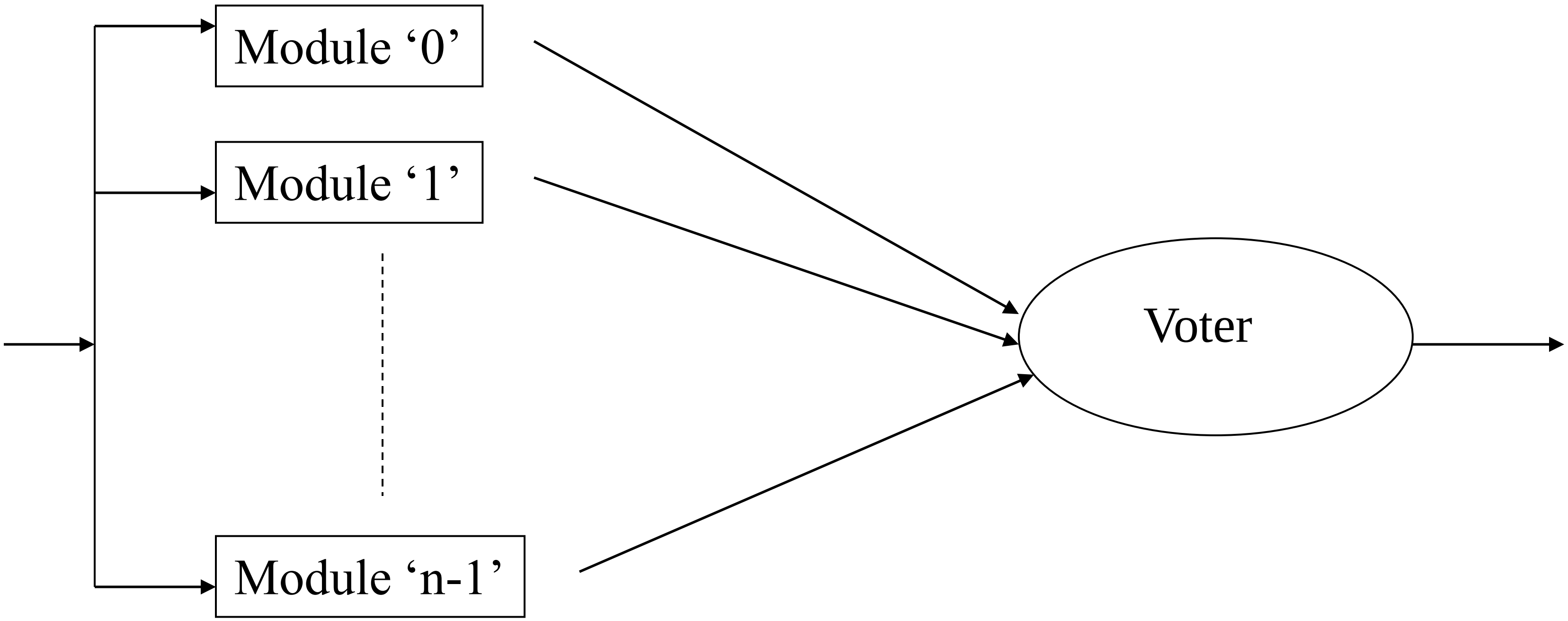
Recovery (fault tolerant) Block concept

```
Recovery Block A

    Acceptance test AT

    Primary block AP

        <Program text>

    Alternate block AQ

        <Program text>
```

Recovery block



N-version programming


Module ‘0’
Module ‘1’
Module ‘n-1’
Voter