Distributed Operating Systems

JAGANNATH UNIVERSITY

Sandeep Kumar Poonia
Head of Dept. CS/IT
B.E., M.Tech., UGC-NET
LM-IAENG, LM-IACSIT,LM-CSTA, LM-AIRCC, LM-SCIEI, AM-UACEE

- Mutual Exclusion
- Election Algorithms
- Atomic Transactions in Distributed Systems

Process Synchronization

Techniques to coordinate execution among processes

- One process may have to wait for another
- Shared resource (e.g. critical section) may require exclusive access

What is *mutual_exclusion?*

- When a process is accessing a shared variable, the process is said to be in a CS (critical section).
- No two process can be in the same CS at the same time. This is called mutual exclusion.

Distributed Mutual Exclusion

- Assume there is agreement on how a resource is identified
  - Pass identifier with requests

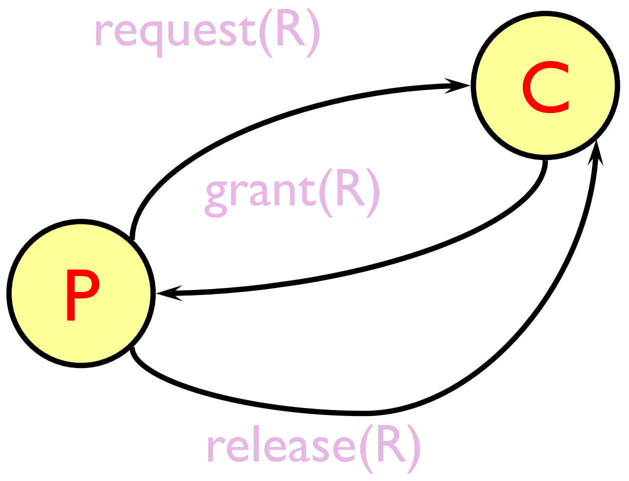
- Create an algorithm to allow a process to obtain exclusive access to a resource

Distributed Mutual Exclusion

- Centralized Algorithm
- Token Ring Algorithm
- Distributed Algorithm
- Decentralized Algorithm

Centralized algorithm

- Mimic single processor system
- One process elected as coordinator



1. **Request** resource
2. Wait for response
3. **Receive grant**
4. *access resource*
5. **Release resource**

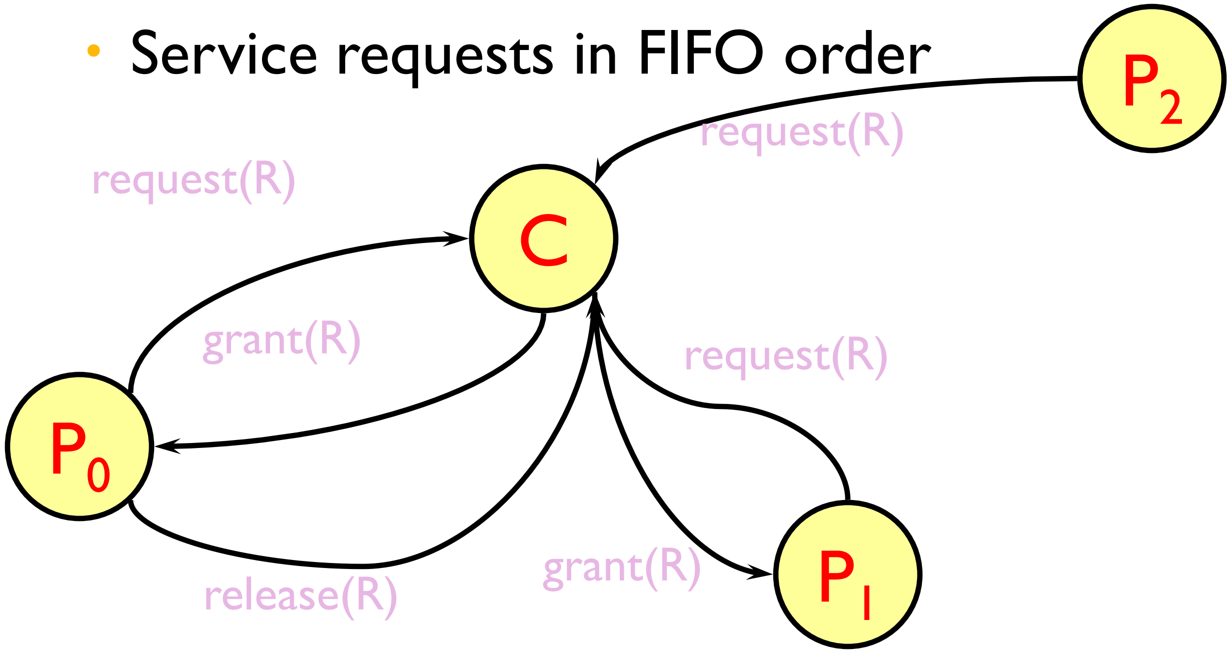
Centralized algorithm

If another process claimed resource:

- Coordinator does not reply until release
- Maintain queue
  - Service requests in FIFO order

<u>Queue</u>

P_1
P_2



ADVANTAGES:

- Mutual exclusion can be achieved.
- The coordinator lets only one process at a time into each CS.
- It is fair.
- Requests are granted in the order in which they are received.
- No process waits forever.
- Easy to implement.
- It can be used for general resource allocation rather than just managing mutual exclusion.

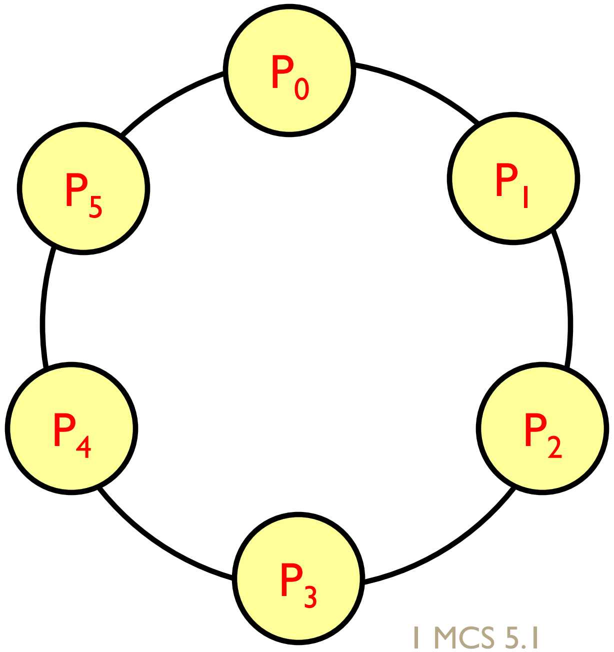
DISADVANTAGES

- The coordinator is a single point of failure, so if it crashes, the entire system may go down.
- If process normally block after making a request, they cannot distinguish a dead coordinator from
  "access denied" since in both cases coordinator doesn't reply.
- In a large system, a single coordinator has to take care of all process.

Token Ring algorithm

Assume known group of processes

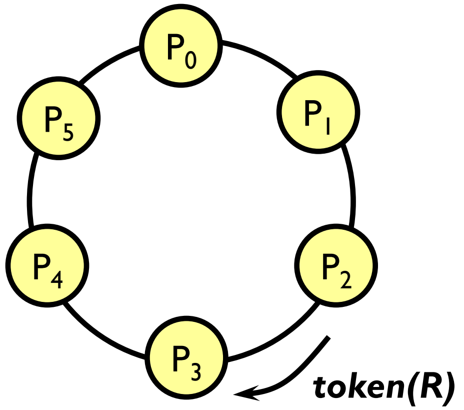
- Some ordering can be imposed on group
- Construct logical ring in software
- Process communicates with neighbor



Token Ring algorithm

- Initialization
  - Process 0 gets token for resource R
- Token circulates around ring

- When process acquires token
  - Checks to see if it needs to enter critical section
  - If no, send token to neighbor
  - If yes, access resource
    - Hold token until done



Token Ring algorithm

- Only one process at a time has token
  - Mutual exclusion guaranteed
- Order well-defined
  - Starvation cannot occur
- If token is lost (e.g. process died)
  - It will have to be regenerated
- Does not guarantee FIFO order
  - sometimes this is undesirable

ADVANTAGES

- The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a CS.
- Since the token circulates among processes in a well-defined order, starvation cannot occur.

DISADVANTAGES

- Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter and leave one critical region.
- If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is not a constant. The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.

DISADVANTAGES…………

- The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases.
- If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can pass the token to the next member down the line.

Ricart & Agrawala algorithm

- Distributed algorithm using reliable multicast and logical clocks
- Process wants to enter critical section:
    - Compose message containing:
        - Identifier (machine ID, process ID)
        - Name of resource
        - Timestamp (totally-ordered Lamport)
    - Send request to all processes in group
    - Wait until everyone gives permission
    - Enter critical section / use resource

Ricart & Agrawala algorithm

- When process receives request:
  - If receiver not interested:
    - Send OK to sender
  - If receiver is in critical section
    - Do not reply; add request to queue
  - If receiver just sent a request as well:
    - Compare timestamps: received & sent messages
    - Earliest wins
    - If receiver is loser, send **OK**
    - If receiver is winner, do not reply, queue
- When done with critical section
  - Send OK to **all queued requests**

Ricart & Agrawala algorithm

- N points of failure
- A lot of messaging traffic
- Demonstrates that a fully distributed algorithm is possible

ADVANTAGES

- Mutual exclusion can be achieved without deadlock or starvation.
- The number of messages required per entry into CS is 2(*n*-1), where the total number of process in the system is *n*.
- *All* processes are involved in *all* decisions.

DISADVANTAGES

- Unfortunately, the single point of failure has been replaced by *n* points of failure.
- If any process crashes, it will fail to respond to requests.
- This silence will be interpreted incorrectly as denial of permission.
- Since the chance of one of the *n* process failing is *n* times the single coordinator failing, we have devised an algorithm which is n times worse than the centralized algorithm.
- Each process must maintain the group membership list itself, including processes entering the group, leaving the group and crashing.
- This algorithm is slower, more complicated, more expensive and less robust than the origianl centralized one.

Lamport's Mutual Exclusion

Each process maintains request queue

- Contains **mutual exclusion requests**

Requesting critical section:

- Process P_i sends request(i, T_i) to all nodes
- Places request on its own queue
- When a process P_j receives a request, it returns a timestamped **ack**

Lamport time

Lamport's Mutual Exclusion

Entering critical section (accessing resource):

- P_i received a message (*ack* or *release*) from every other process with a timestamp larger than T_i
- P_i's request has the earliest timestamp in its queue

Difference from Ricart-Agrawala:

- Everyone responds … always - no hold-back
- Process decides to go based on whether its request is the earliest in its queue

Lamport's Mutual Exclusion

Releasing critical section:

- Remove request from its own queue
- Send a timestamped *release* message

- When a process receives a *release* message
  - Removes request for that process from its queue
  - This may cause its own entry have the earliest timestamp in the queue, enabling it to access the critical section

Characteristics of Decentralized Algorithms

- No machine has complete information about the system state
- Machines make decisions based only on local information
- Failure of one machine does not ruin the algorithm
- Three is no implicit assumption that a global clock exists

Decentralized Algorithm

- Based on the Distributed Hash Table (DHT) system structure previously introduced
  - Peer-to-peer
  - Object names are hashed to find the successor node that will store them
- Here, we assume that *n* replicas of each object are stored

Placing the Replicas

- The resource is known by a unique name: *rname*
  - Replicas: *rname-0, rname-l, …, rname-(n-1)*
  - *rname-i* is stored at *succ(rname-i)*, where names and site names are hashed as before
  - If a process knows the name of the resource it wishes to access, it also can generate the hash keys that are used to locate all the replicas

The Decentralized Algorithm

- Every replica has a coordinator that controls access to it (the coordinator is the node that stores it)
- For a process to use the resource it must receive permission from $m > n/2$ coordinators
- This guarantees exclusive access as long as a coordinator only grants access to one process at a time

The Decentralized Algorithm

- The coordinator notifies the requester when it has been denied access as well as when it is granted
    - Requester must "count the votes", and decide whether or not overall permission has been granted or denied
- If a processor (requester) gets fewer than *m* votes it will wait for a random time and then ask again

Analysis

- If a resource is in high demand, multiple requests will be generated
- It's possible that processes will wait a long time to get permission
- Deadlock?
- Resource usage drops

Analysis

- More robust than the central coordinator approach. If one coordinator goes down others are available.

    - If a coordinator fails and resets then it will not remember having granted access to one requestor, and may then give access to another. According to the authors, it is highly unlikely that this will lead to a violation of mutual exclusion.

COMPARISON

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

- A comparison of three mutual exclusion algorithms.

Election Algorithms

- If we are using one process as a coordinator for a shared resource …
- …how do we select that one process?

Solution – an *Election*

- All nodes currently involved get together to *choose* a coordinator
- If the coordinator crashes or becomes isolated, elect a new coordinator
- If a previously crashed or isolated node, comes on line, a new election *may* have to be held

Election Algorithms

- Wired systems
  - Bully algorithm
  - Ring algorithm
- Wireless systems
- Very large-scale systems

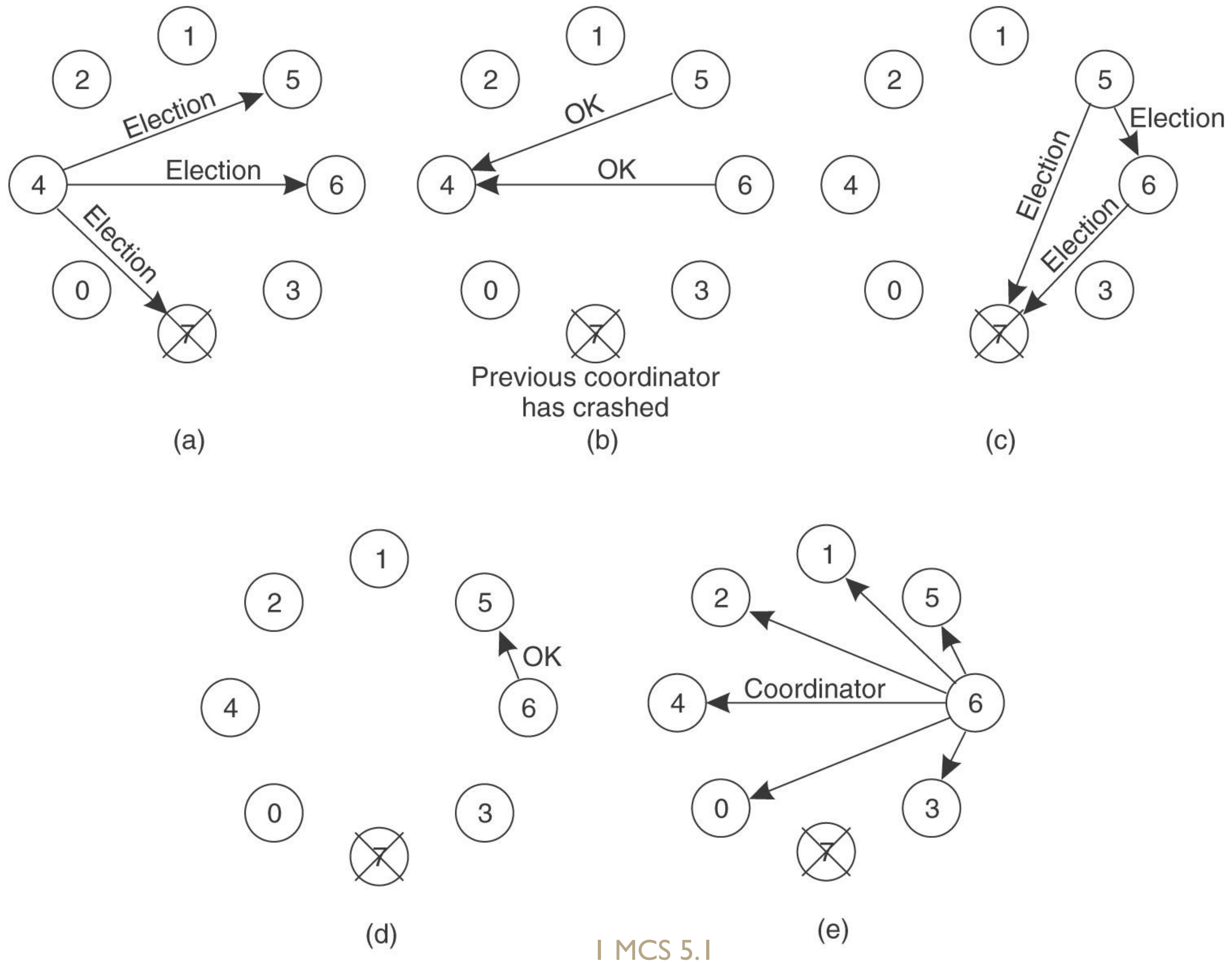
Bully Algorithm

- Assume
    - All processes know about each other
    - Processes numbered uniquely
    - They do not know each other’s state
- Suppose *P* notices no coordinator
    - Sends *election message* to all higher numbered processes
    - If none response, *P* takes over as coordinator
    - If any responds, *P* yields
- …

Bully Algorithm (continued)

- …
- Suppose *Q* receives *election message*
  - Replies *OK* to sender, saying it will take over
  - Sends a new *election message* to higher numbered processes
- Repeat until only one process left standing
  - Announces victory by sending message saying that it is the coordinator

Bully Algorithm (continued)



Bully Algorithm (continued)

- …
- Suppose *R* comes back on line
    - Sends a new *election message* to higher numbered processes
- Repeat until only one process left standing
    - Announces victory by sending message saying that it is the coordinator (if not already the coordinator)
- Existing (lower numbered) coordinator yields
    - Hence the term "bully"

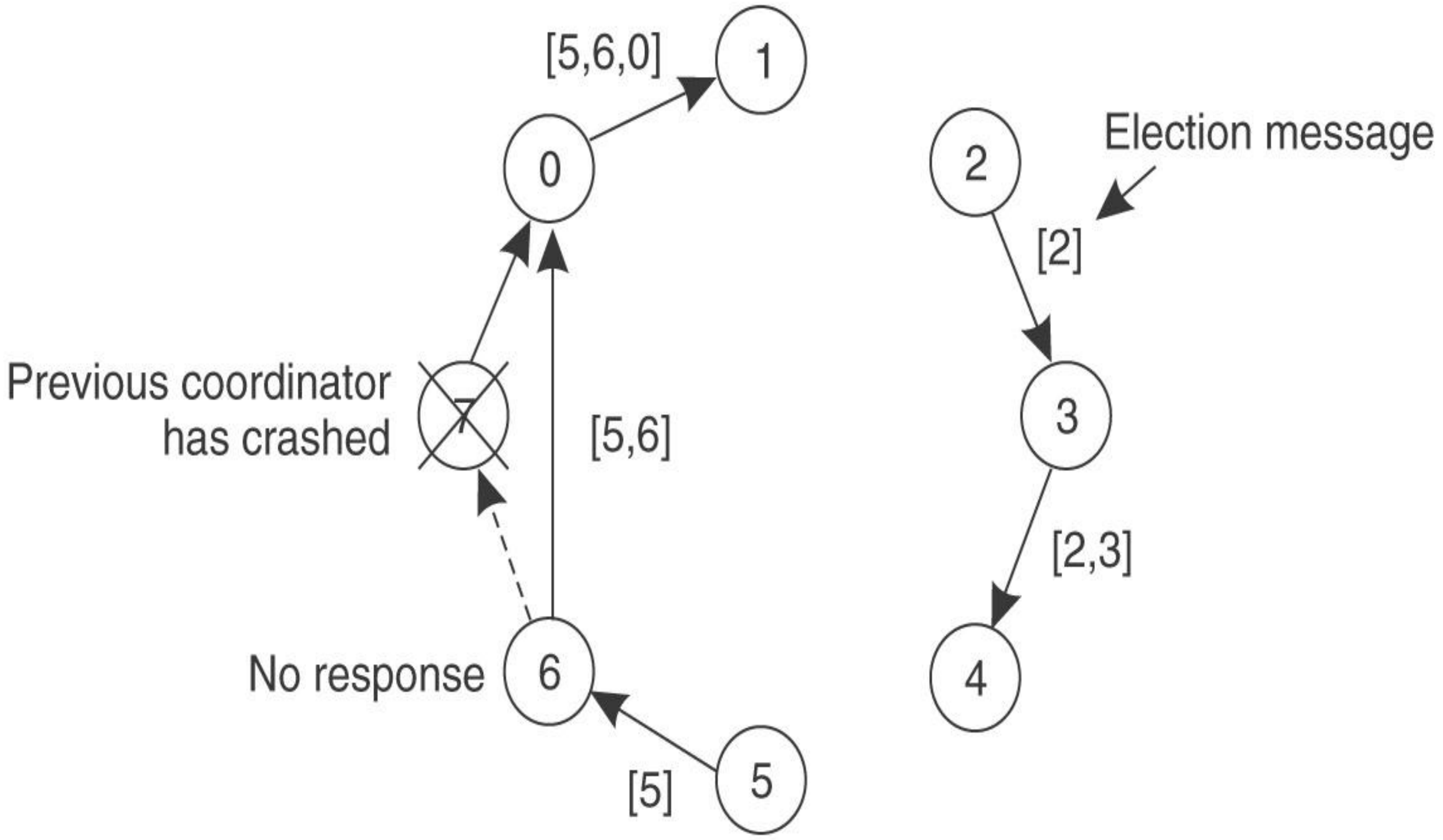
Alternative – Ring Algorithm

- All processes organized in ring
  - Independent of process number
- Suppose *P* notices no coordinator
  - Sends *election message* to successor with own process number in body of message
  - (If successor is down, skip to next process, etc.)
- Suppose *Q* receives an election message
  - Adds own process number to list in message body
- …

Alternative – Ring Algorithm

- Suppose *P* receives an election message with its own process number in body
    - Changes message to *coordinator* message, preserving body
    - All processes recognize highest numbered process as new coordinator
- If multiple messages circulate …
    - …they will all contain same list of processes (eventually)
- If process comes back on-line
    - Calls new election

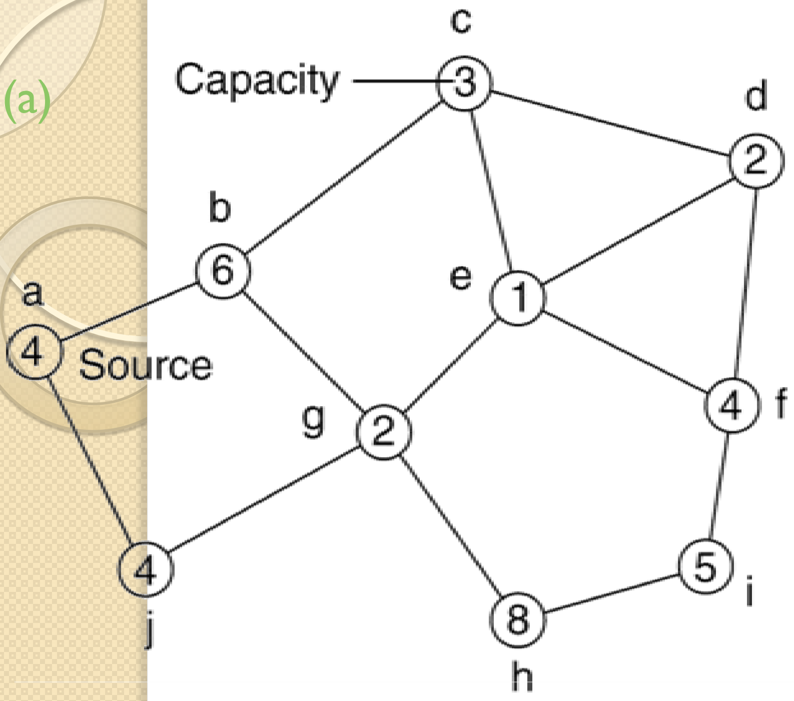
Ring Algorithm (continued)



Wireless Environments

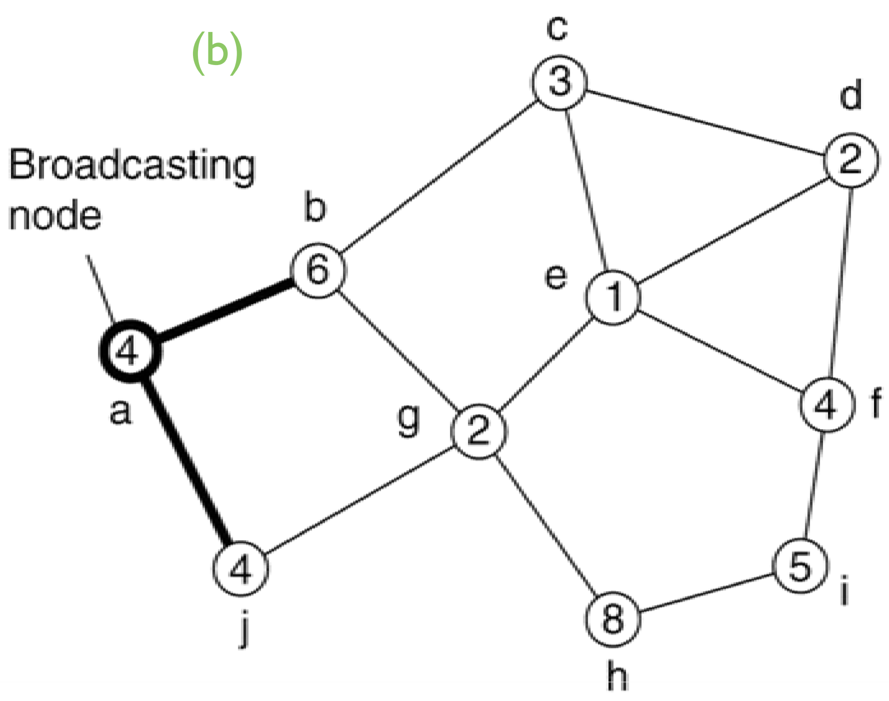
- Unreliable, and nodes may move
- Network topology constantly changing
- Algorithm:
  1. Any node starts by sending out an ELECTION message to neighbors
  2. When a node receives an ELECTION message for the first time, it forwards to neighbors, and designates the sender as its parent
  3. It then waits for responses from is neighbors
     - Responses may carry resource information
  4. When a node receives an ELECTION message for the second time, it just OKs it

(a)



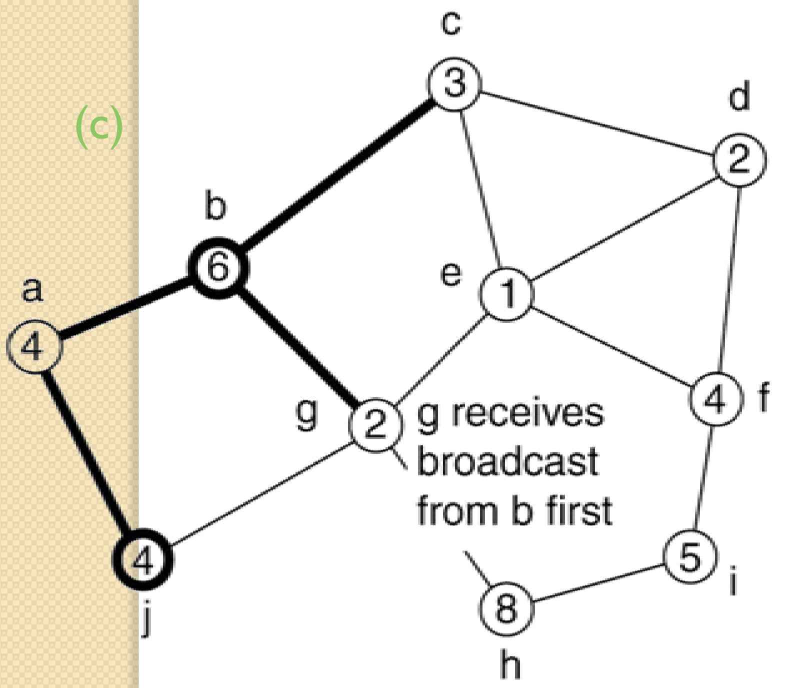
Capacity
Source
a
b
c
d
e
f
g
h
i
j
4
6
3
2
1
4
2
4
8
5

(b)



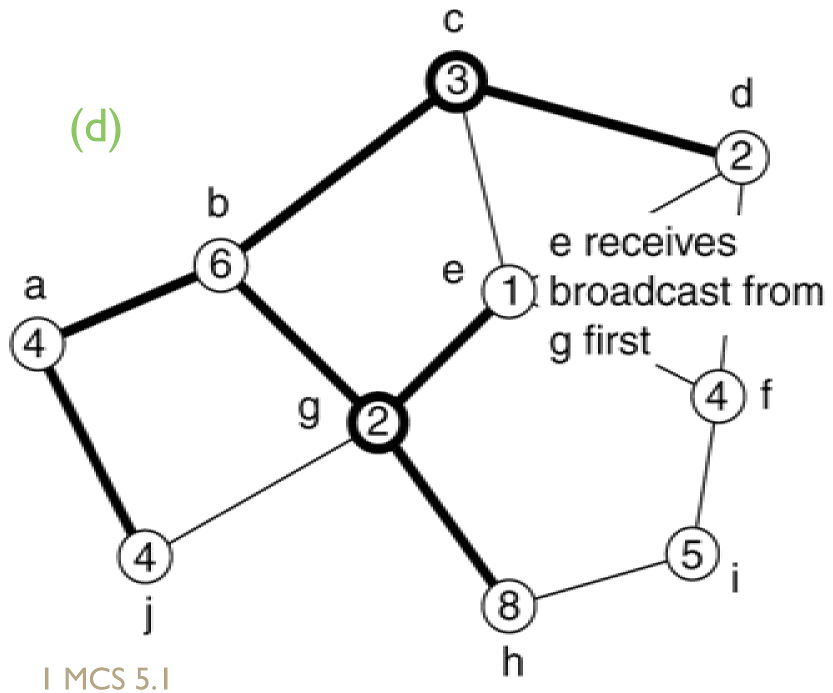
Broadcasting node
a
b
c
d
e
f
g
h
i
j

(c)



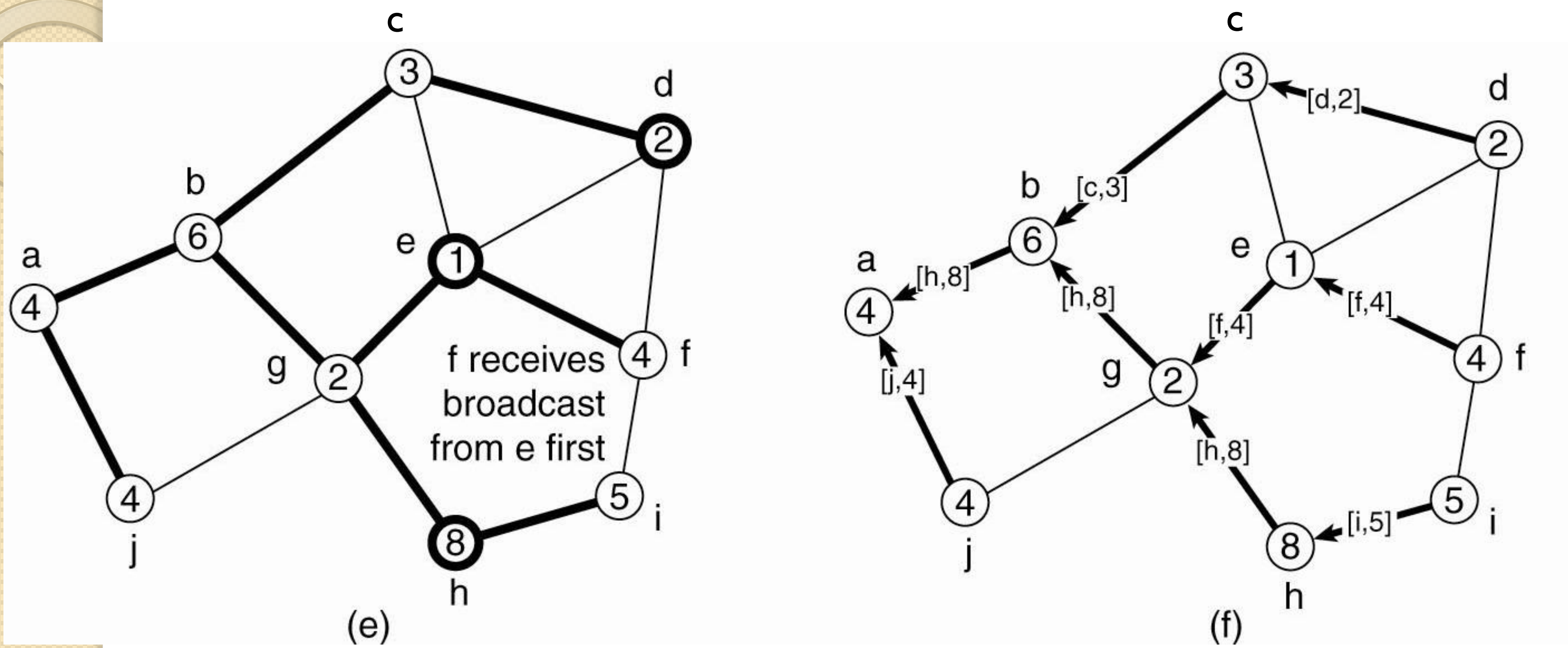
g receives broadcast from b first
a
b
c
d
e
f
g
h
i
j

(d)



e receives broadcast from g first
a
b
c
d
e
f
g
h
i
j

Elections in Wireless Environment



(e) The build-tree phase. (f) Reporting of best node to source.

Very Large Scale Networks

- Sometimes more than one node should be selected
- Nodes organized as peers and *super-peers*
  - Elections held within each peer group
  - Super-peers coordinate among themselves

Atomic Transaction

Transaction concepts

- OS Processes ≅ DBMS Transactions
- A collection of actions that make consistent transformation of system states while preserving consistency
- Termination of transactions – *commit* vs *abort*
- Example:

  T : x = x + y

  Read(x)
  Read(y)
  Write(x)
  Commit

$$\begin{tikzcd}
R(x) \arrow[dr] & & \\
& W(x) \arrow[r] & C \\
R(y) \arrow[ur] & &
\end{tikzcd}$$

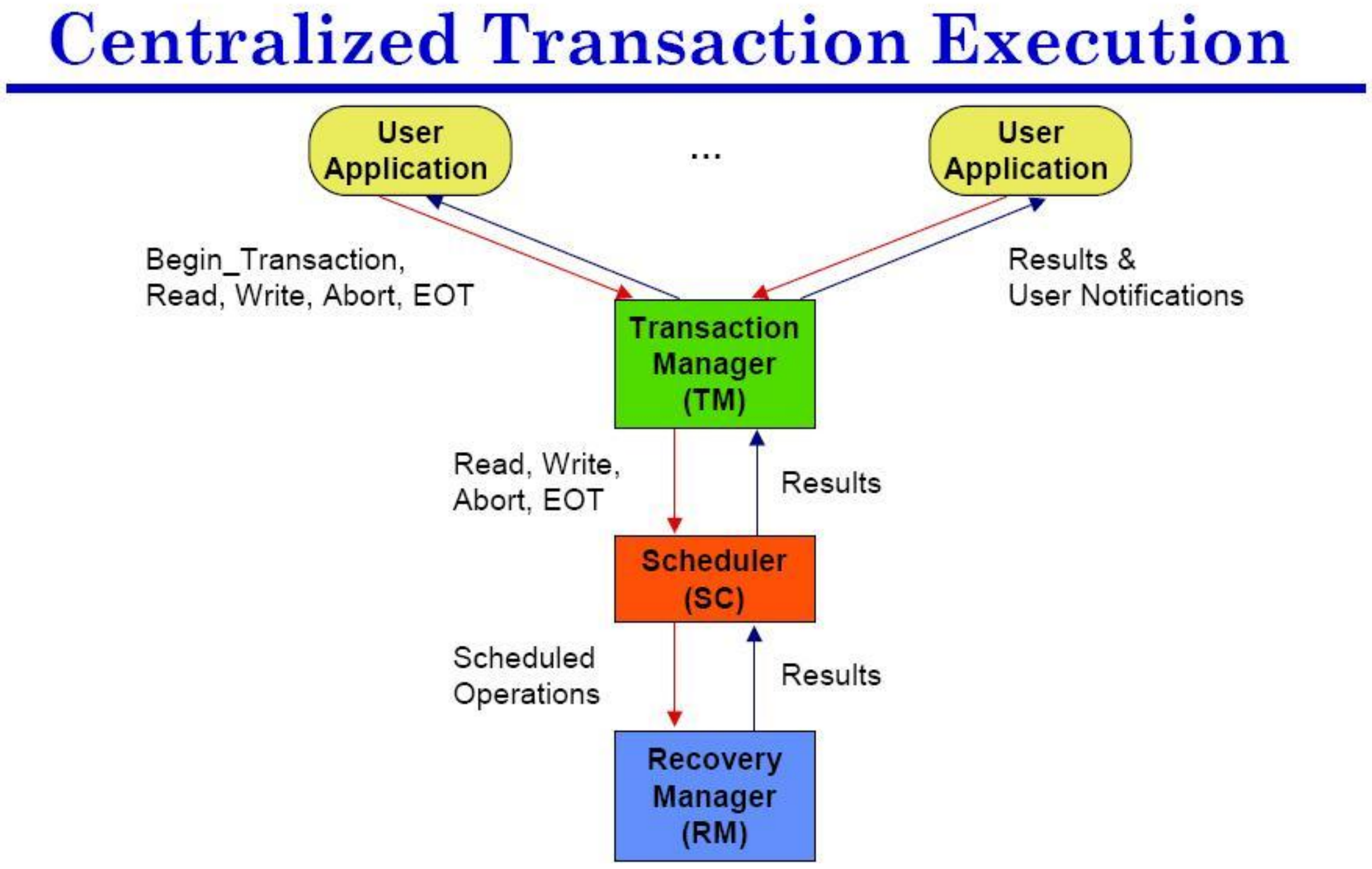
Definition – *Transaction*

- A sequence of operations that perform a single logical function
- Examples
  - Withdrawing money from your account
  - Making an airline reservation
  - Making a credit-card purchase
  - Registering for a course
  - …
- Usually used in context of databases

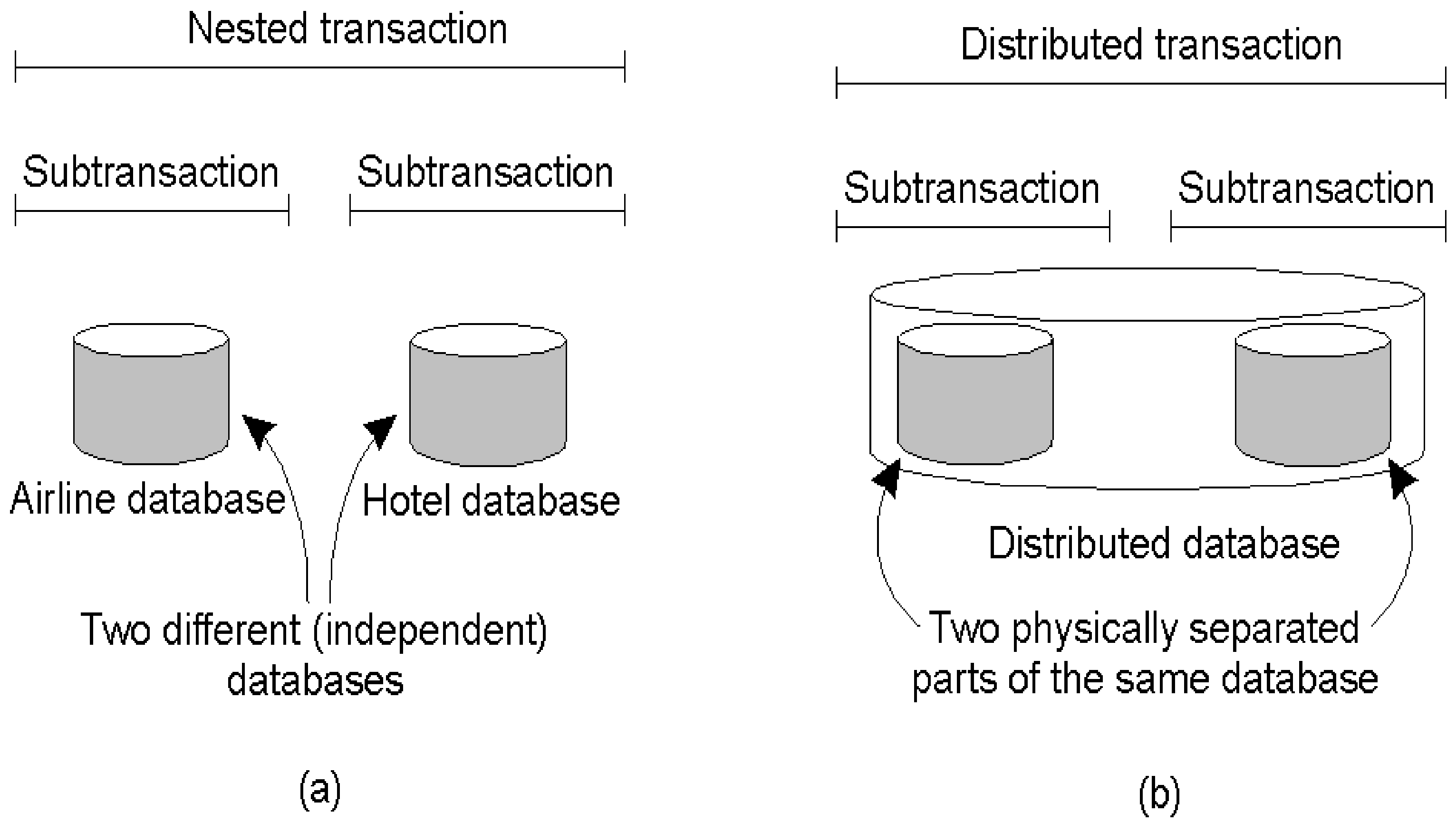
Transaction Processing Issues

- Transaction structure
  - Flat vs Nested vs Distributed
- Internal database consistency
  - Semantic data control and integrity enforcement
- Reliability protocols
  - Atomicity and durability
  - Local recovery protocols
  - Global commit protocols
- Concurrency control algorithms
- Replica control protocols

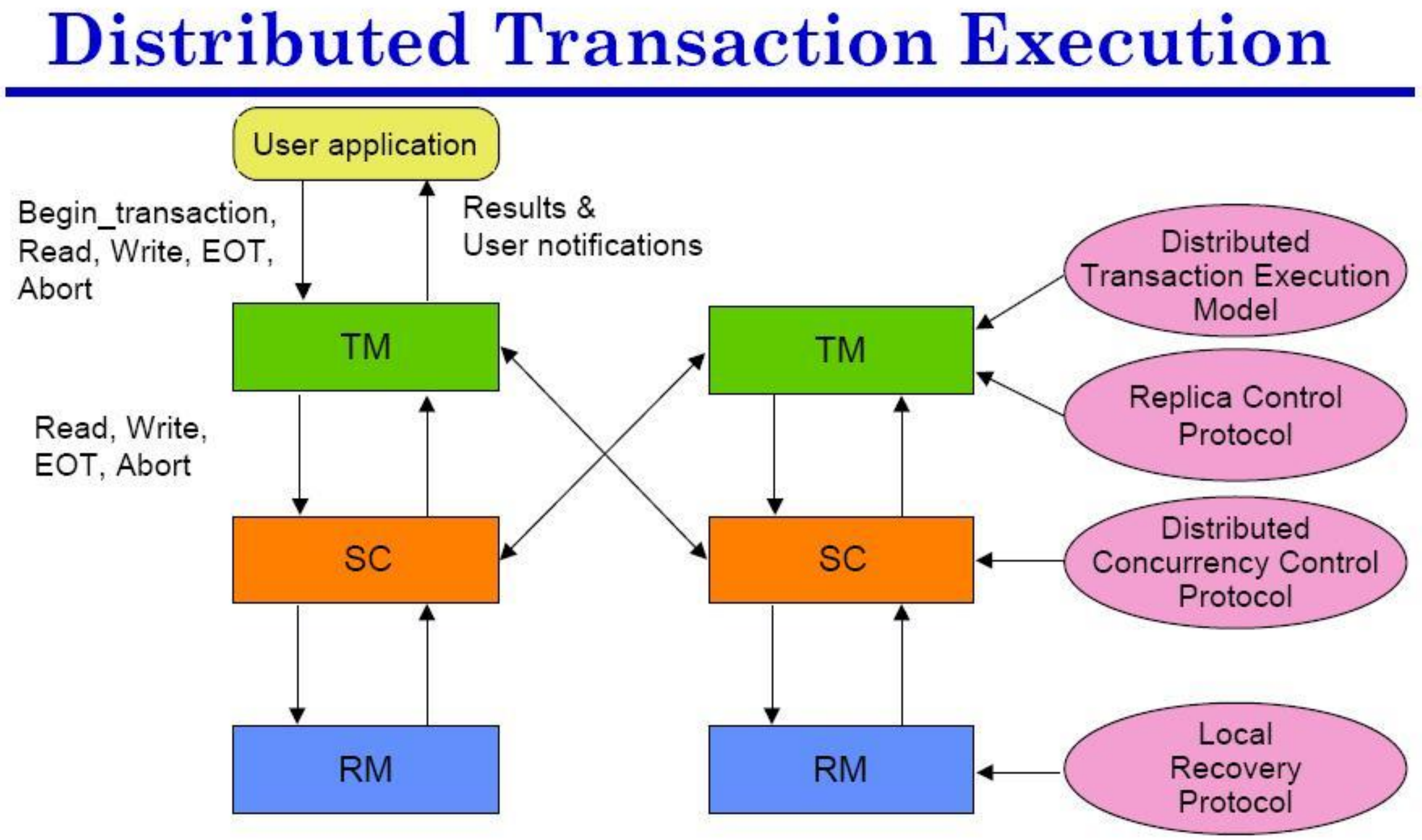
Transaction execution



Nested vs Distributed Transactions



Distributed Transaction execution


Distributed Transaction Execution
User application
Begin_transaction, Read, Write, EOT, Abort
Results & User notifications
TM
TM
Read, Write, EOT, Abort
SC
SC
RM
RM
Distributed Transaction Execution Model
Replica Control Protocol
Distributed Concurrency Control Protocol
Local Recovery Protocol

Definition – *Atomic Transaction*

- A transaction that happens completely or not at all
  - No partial results
- Example:
  - Cash machine hands you cash and deducts amount from your account
  - Airline confirms your reservation and
    - Reduces number of free seats
    - Charges your credit card
    - (Sometimes) increases number of meals loaded on flight
  - …

Atomic Transaction Review

- Fundamental principles – *A C I D*
  - *Atomicity* – to outside world, transaction happens indivisibly
  - *Consistency* – transaction preserves system invariants
  - *Isolated* – transactions do not interfere with each other
  - *Durable* – once a transaction "commits," the changes are permanent

Programming in a Transaction System

- *Begin_transaction*
  - Mark the start of a transaction
- *End_transaction*
  - Mark the end of a transaction and try to "commit"
- *Abort_transaction*
  - Terminate the transaction and restore old values
- *Read*
  - Read data from a file, table, etc., on behalf of the transaction
- *Write*
  - Write data to file, table, etc., on behalf of the transaction

Programming in a Transaction System (continued)

- As a matter of practice, separate transactions are handled in separate threads or processes
- *Isolated* property means that two concurrent transactions are *serialized*
    - I.e., they run in some indeterminate order with respect to each other

Programming in a Transaction System (continued)

- *Nested Transactions*
  - One or more transactions inside another transaction
  - May individually commit, *but* may need to be undone
- Example
  - Planning a trip involving three flights
  - Reservation for each flight "commits" individually
  - Must be undone if entire trip cannot commit

Tools for Implementing Atomic Transactions (single system)

- Stable storage
  - i.e., write to disk "atomically"
- Log file
  - i.e., record actions in a log before "committing" them
  - Log in stable storage
- Locking protocols
  - Serialize *Read* and *Write* operations of same data by separate transactions
- …

Tools for Implementing Atomic Transactions (continued)

- *Begin_transaction*
  - Place a *begin* entry in log
- *Write*
  - Write updated data to log
- *Abort_transaction*
  - Place *abort* entry in log
- *End_transaction* (i.e., *commit*)
  - Place *commit* entry in log
  - Copy logged data to files
  - Place *done* entry in log

Tools for Implementing Atomic Transactions (continued)

- Crash recovery – search log
- If *begin* entry, look for matching entries
- If *done*, do nothing (all files have been updated)
- If *abort*, undo any permanent changes that transaction may have made
- If *commit* but not *done*, copy updated blocks from log to files, then add *done* entry

Distributed ACID

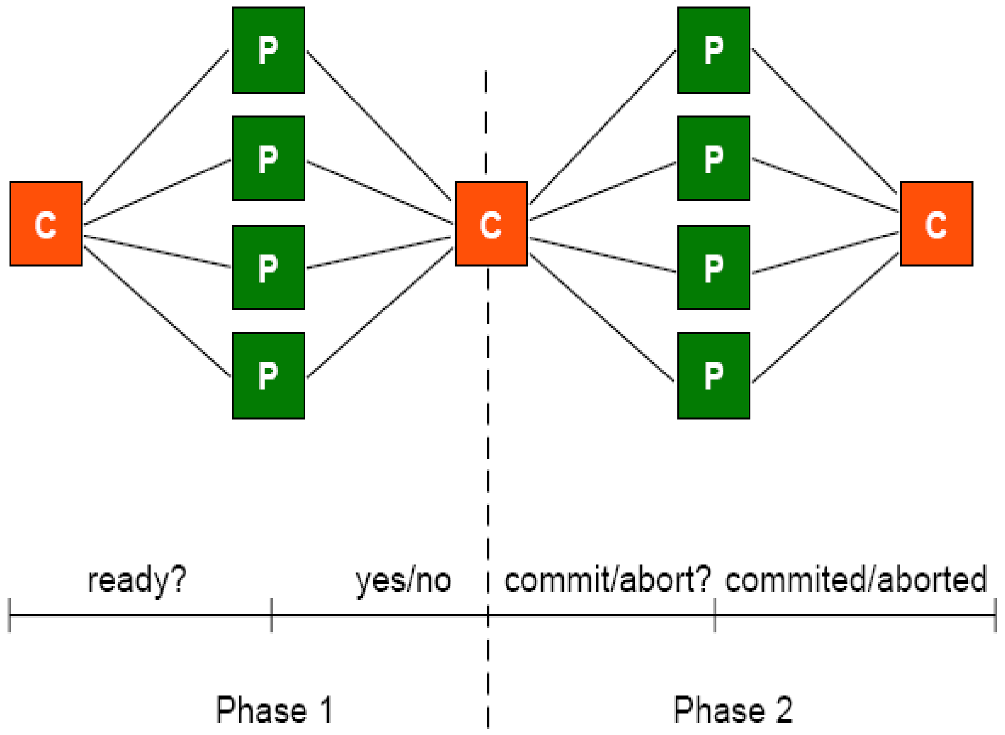
- *Global Atomicity:* All sub transactions of a distributed transaction must commit or all must abort.
  - An *atomic commit protocol*, initiated by a *coordinator* (*e.g.*, the transaction manager), ensures this.
  - Coordinator must poll *cohorts* to determine if they are all willing to commit.
- *Global deadlocks*: there must be no deadlocks involving multiple sites
- *Global serialization*: distributed transaction must be globally serializable

Distributed Atomic Transactions

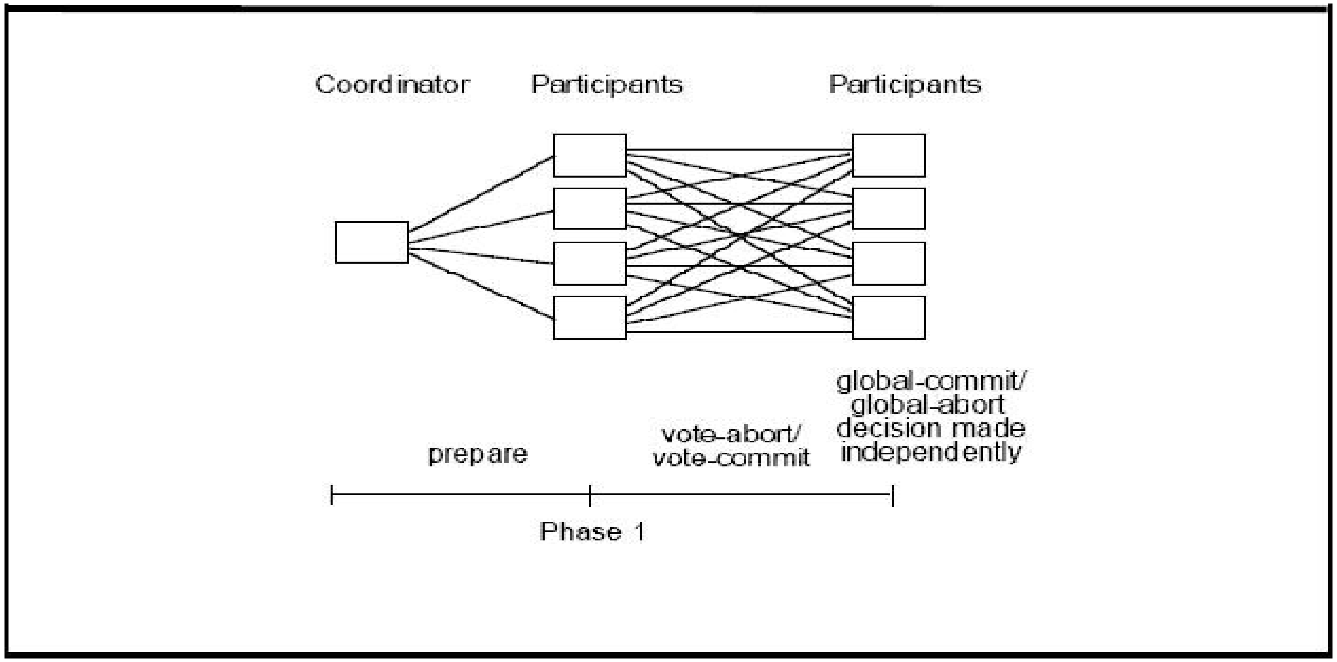
- Atomic transactions that span multiple sites and/or systems
- Same semantics as atomic transactions on single system
  - *A C I D*
- Failure modes
  - Crash or other failure of one site or system
  - Network failure or partition
  - Byzantine failures

2PC phases

Centralized 2PC



Distributed 2PC



- The Coordinator initiates 2PC
- The participants run a distributed algorithm to reach the agreement of global commit or abort.

Two-Phase Commit

- One site is elected *coordinator* of the transaction *T*
  - Using *Election* algorithms
- *Phase 1:* When coordinator is ready to commit the transaction
  - Place *Prepare*(*T*) state in log on stable storage
  - Send *Vote_request*(*T*) message to all other participants
  - Wait for replies

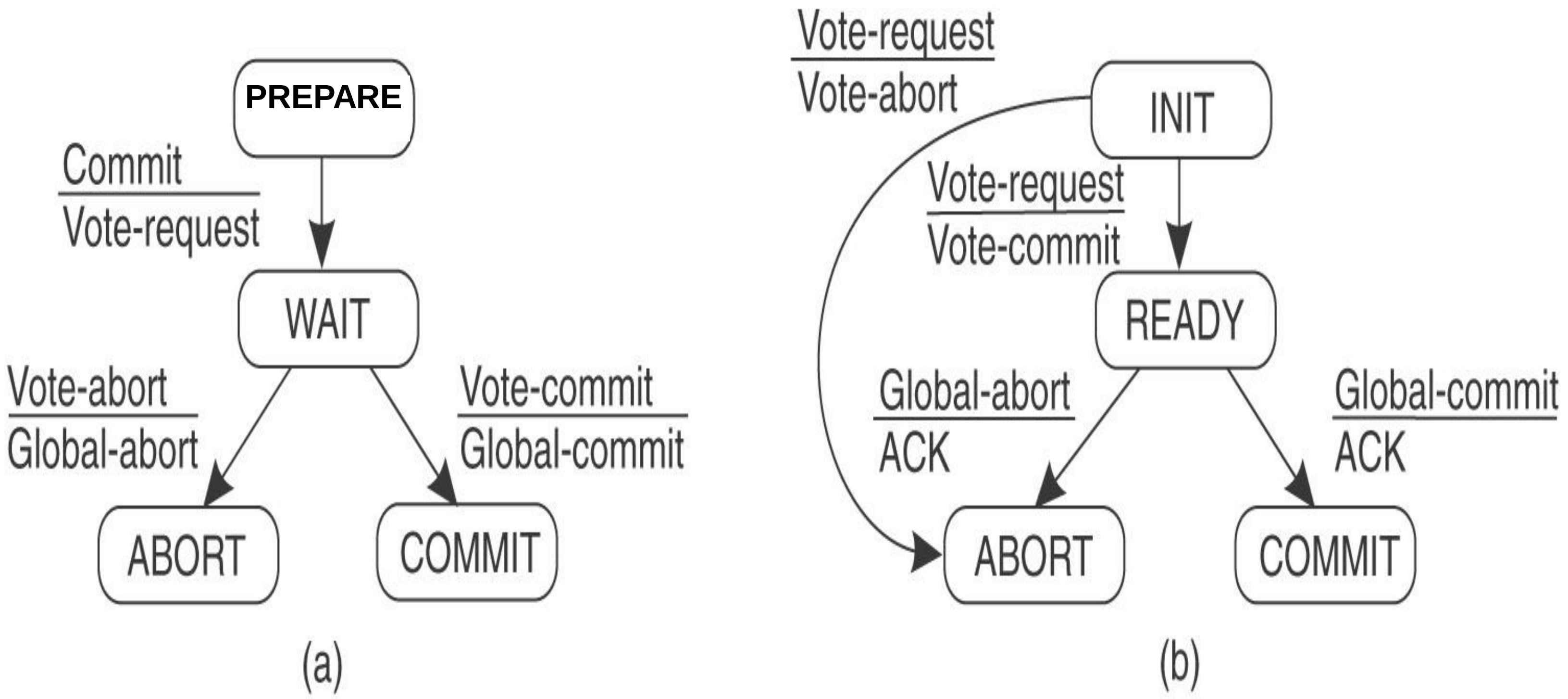
Two-Phase Commit (continued)

- *Phase 2:* Coordinator
  - If any participant replies *Abort*(*T*)
    - Place *Abort*(*T*) state in log on stable storage
    - Send *Global_Abort*(*T*) message to all participants
    - Locally abort transaction *T*
  - If *all* participants reply *Ready_to_commit*(*T*)
    - Place *Commit*(*T*) state in log on stable storage
    - Send *Global_Commit*(*T*) message to all participants
    - Proceed to commit transaction locally

Two-Phase Commit (continued)

- Phase I: Participant gets *Vote_request*(*T*) from coordinator
  - Place *Abort*(*T*) or *Ready*(T) state in local log
  - Reply with *Abort*(*T*) or *Ready_to_commit*(*T*) message to coordinator
  - If *Abort*(*T*) state, locally abort transaction
- Phase II: Participant
  - Wait for *Global_Abort*(*T*) or *Global_Commit*(*T*) message from coordinator
  - Place *Abort*(*T*) or *Commit*(T) state in local log
  - Abort or commit locally per message

Two-Phase Commit States



Failure Recovery – Two-Phase Commit

- Failure modes (from coordinator's point of view)
  - Own crash
  - *Wait* state: No response from some participant to *Vote_request* message
- Failure modes (from participant's point of view)
  - Own crash
  - *Ready* state: No message from coordinator to *Global_Abort*(*T*) or *Global_Commit*(*T*)

Lack of Response to Coordinator *Vote_Request*(*T*) message

- E.g.,
  - participant crash
  - Network failure
- Timeout is considered equivalent to *Abort*
  - Place *Abort*(*T*) state in log on stable storage
  - Send *Global_Abort*(*T*) message to all participants
  - Locally abort transaction *T*

Coordinator Crash

- Inspect Log
- If *Abort* or *Commit* state
  - Resend corresponding message
  - Take corresponding local action
- If *Prepare* state, either
  - Resend *Vote_request*(*T*) to all other participants and wait for their responses; *or*
  - Unilaterally abort transaction
    - I.e., put *Abort*(*T*) in own log on stable store
    - Send *Global_Abort*(*T*) message to all participants
- If nothing in log, abort transaction as above

No Response to Participant's *Ready_to_commit*(*T*) message

- Re-contact coordinator, ask what to do
- If unable to contact coordinator, contact other participants, ask if they know
- If any other participant is in *Abort* or *Commit* state
    - Take equivalent action
- Otherwise, wait for coordinator to restart!
    - Participants are blocked, unable to go forward or back
    - Frozen in *Ready* state!

Participant Crash

- Inspect local log
  - *Commit* state:
    - Redo/replay the transaction
  - *Abort* state:
    - Undo/abort the transaction
  - No records about *T*:
    - Same as *local_abort*(*T*)
  - *Ready* State:
    - Same as no response to *Ready_to_commit*(*T*) message

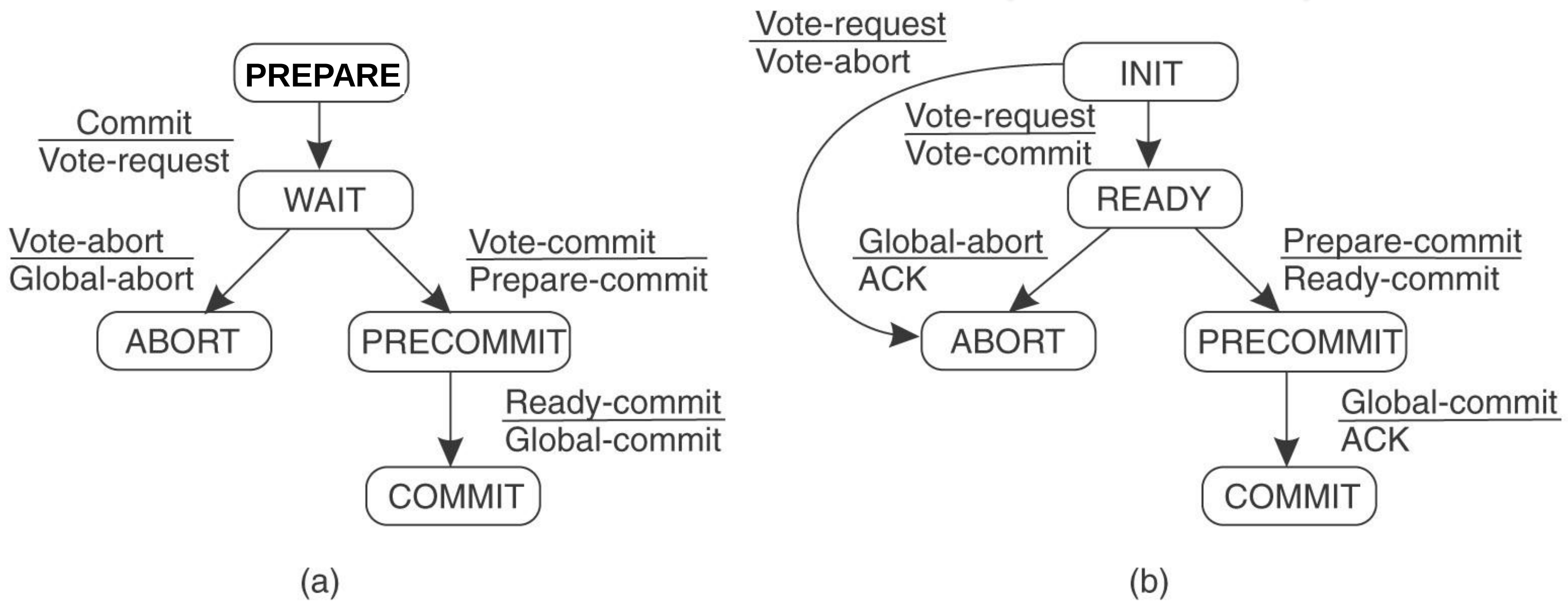
Two-Phase Commit Summary

- Widely used in distributed transaction and database systems
- Generally works well
  - When coordinators are likely to reboot quickly
  - When network partition is likely to end quickly
- Still subject to participant blocking

Three-Phase Commit

- Minor variation
- Widely quoted in literature
- Rarely implemented
  - Because indefinite blocking due to coordinator failures doesn't happen very often in real life!

Three-Phase Commit (continued)


PREPARE
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Prepare-commit
ABORT
PRECOMMIT
Ready-commit / Global-commit
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Prepare-commit / Ready-commit
ABORT
PRECOMMIT
Global-commit / ACK
COMMIT
(b)

- There is no state from which a transition can be made to either *Commit* or *Abort*
- There is no state where it is not possible to make a final decision and from which transition can be made to *Commit*.

Three-Phase Commit (continued)

- Coordinator sends *Vote_Request* (as before)
- If all participants respond affirmatively,
  - Put *Precommit* state into log on stable storage
  - Send out *Prepare_to_Commit* message to all
- After all participants acknowledge,
  - Put *Commit* state in log
  - Send out *Global_Commit*

Three-Phase Commit Failures

- Coordinator blocked in *Ready* state
  - Safe to abort transaction
- Coordinator blocked in *Precommit* state
  - Safe to issue *Global_Commit*
  - Any crashed or partitioned participants will commit when recovered
- …

Three-Phase Commit Failures (continued)

- Participant blocked in *Precommit* state
  - Contact others
  - Collectively decide to commit
- Participant blocked in *Ready* state
  - Contact others
  - If any in *Abort*, then abort transaction
  - If any in *Precommit*, the move to *Precommit* state
  - …

Three-Phase Commit Summary

- If any processes are in *Precommit* state, then all crashed processes will recover to
  - *Ready*, *Precommit*, or *Committed* states
- If any process is in *Ready* state, then all other crashed processes will recover to
  - *Init*, *Abort*, or *Precommit*
  - Surviving processes can make collective decision