# 7. Transaction Management for distributed databases

# 7. Transaction Management for distributed databases

# 7. Transaction Management for distributed databases

# 7. Transaction Management for distributed databases

# 7. Transaction Management for distributed databases

# Distributed Transactions

- In distributed DBS, transactions across several nodes
- Commit as an atomic event $\rightarrow$ Simultaneous in distributed nodes
- Distributed synchronization in order to guarantee consistency in interleaved executions
- Deadlock detection

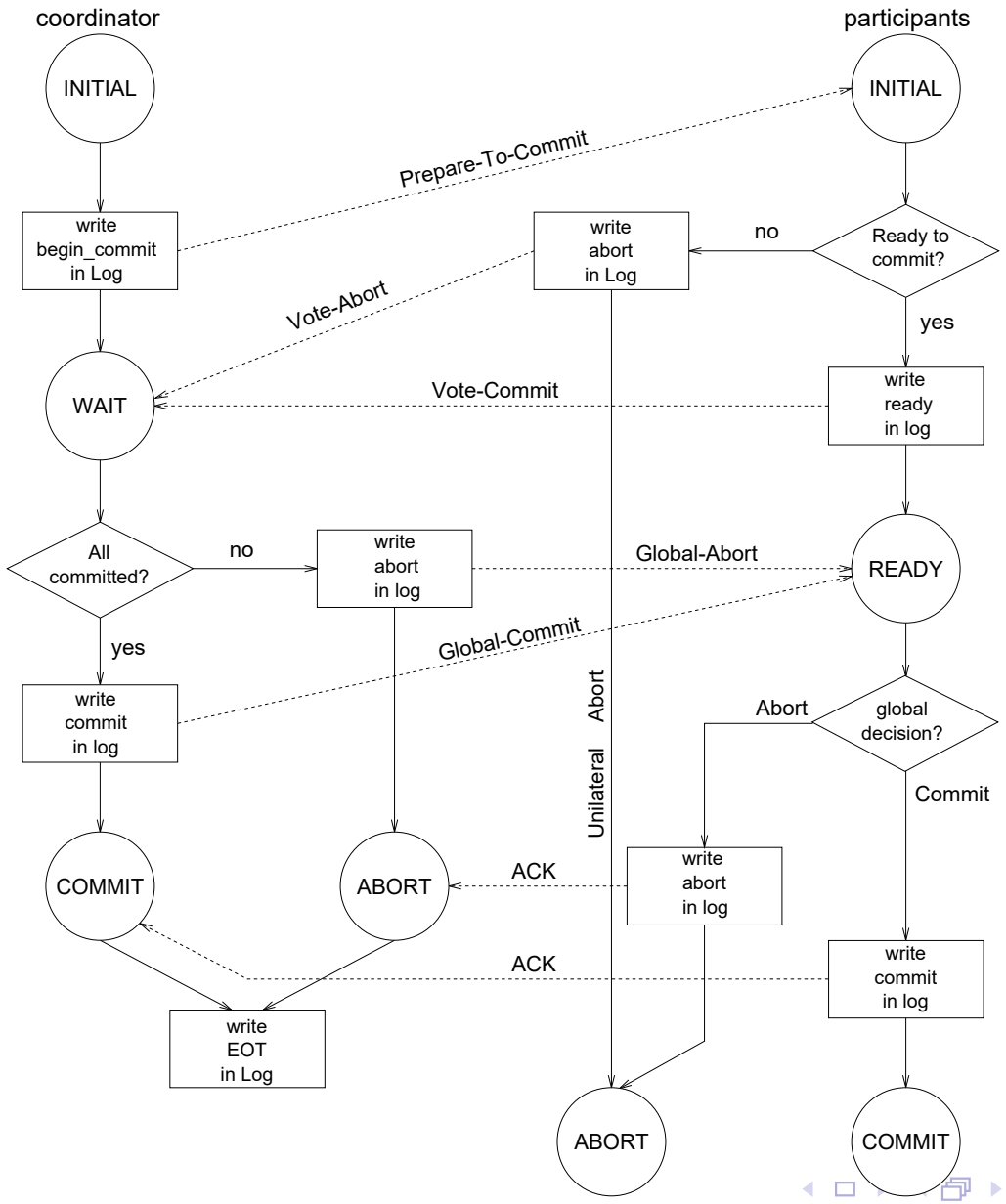
# Requirements for distributed commit

- Commit protocol: Guarantee for atomicity and durability
- Requirements for distributed cases
  - All nodes make a decision (**`Commit`**, **`Abort`**); globally, all nodes make the same decision
  - **`Commit`** only if all nodes vote “yes“
  - If no failure occurs and all nodes vote “yes“ $\rightsquigarrow$ global decision is **`commit`**
  - All processes terminate

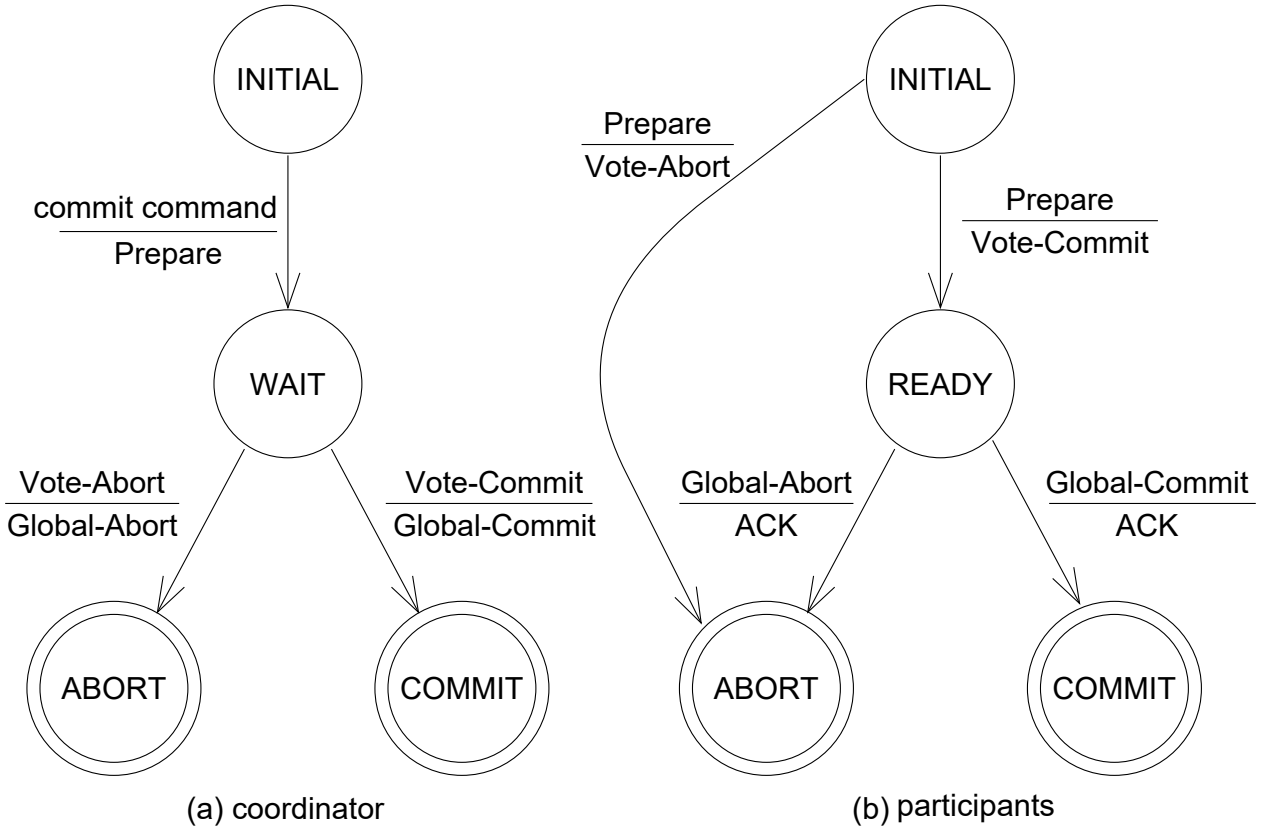
# Two-phase commit protocol

- Roles: 1 coordinator, several participants
- Execution:
  1. *Voting phase*
     1. Coordinator asks participants whether **`Commit`** can be performed
     2. Participants signal their decision to coordinator
  2. *Decision phase*
     1. Coordinator makes a decision based on participants' signals (all **`commit`** $\rightarrow$ **`Global-Commit`**; one **`Abort`** $\rightarrow$ **`Global-Abort`**
     2. Participants that voted "yes" wait for decision

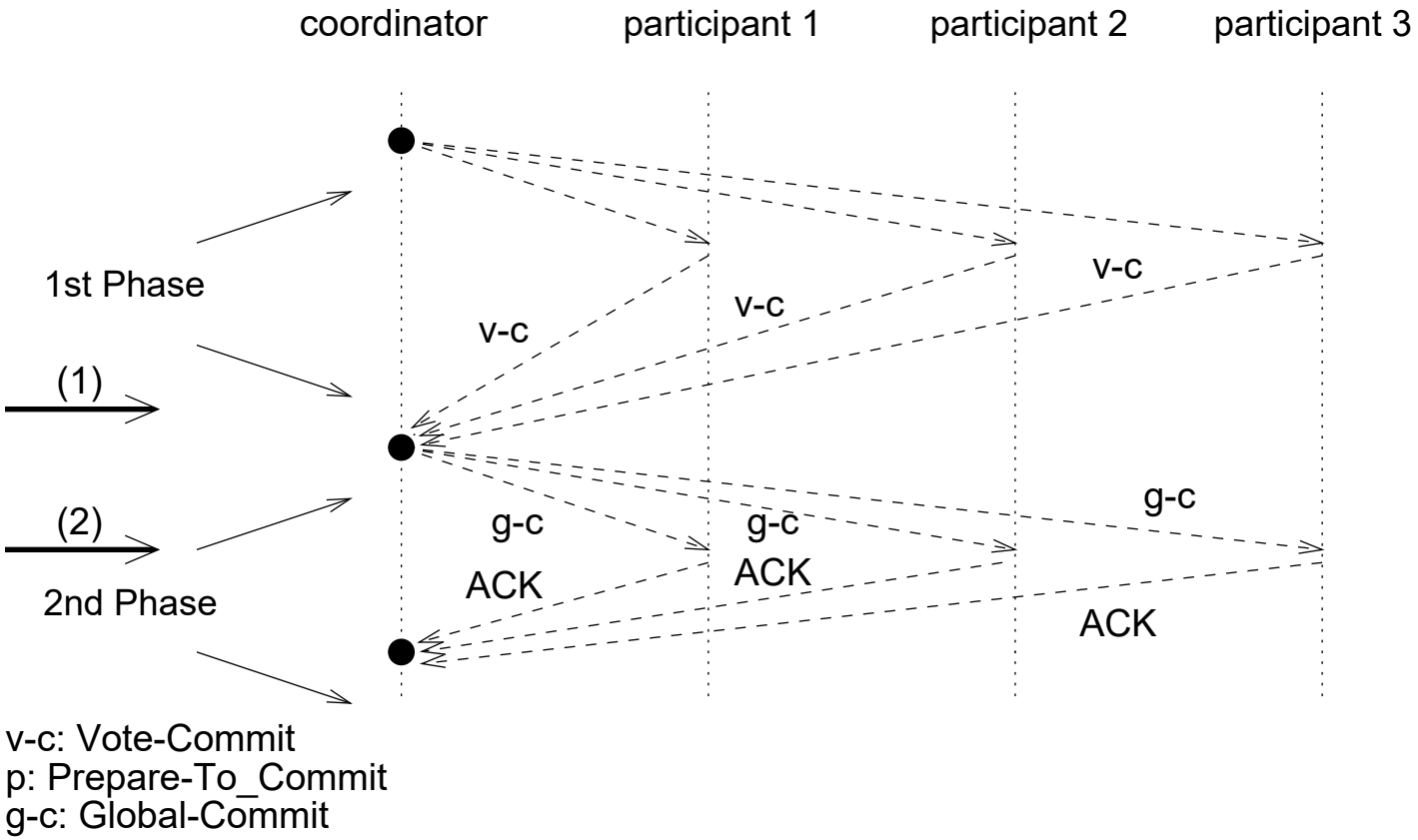
# 2PC: Execution scheme



# 2PC: State transition



# 2PC: Problems I



# 2PC: Problems II

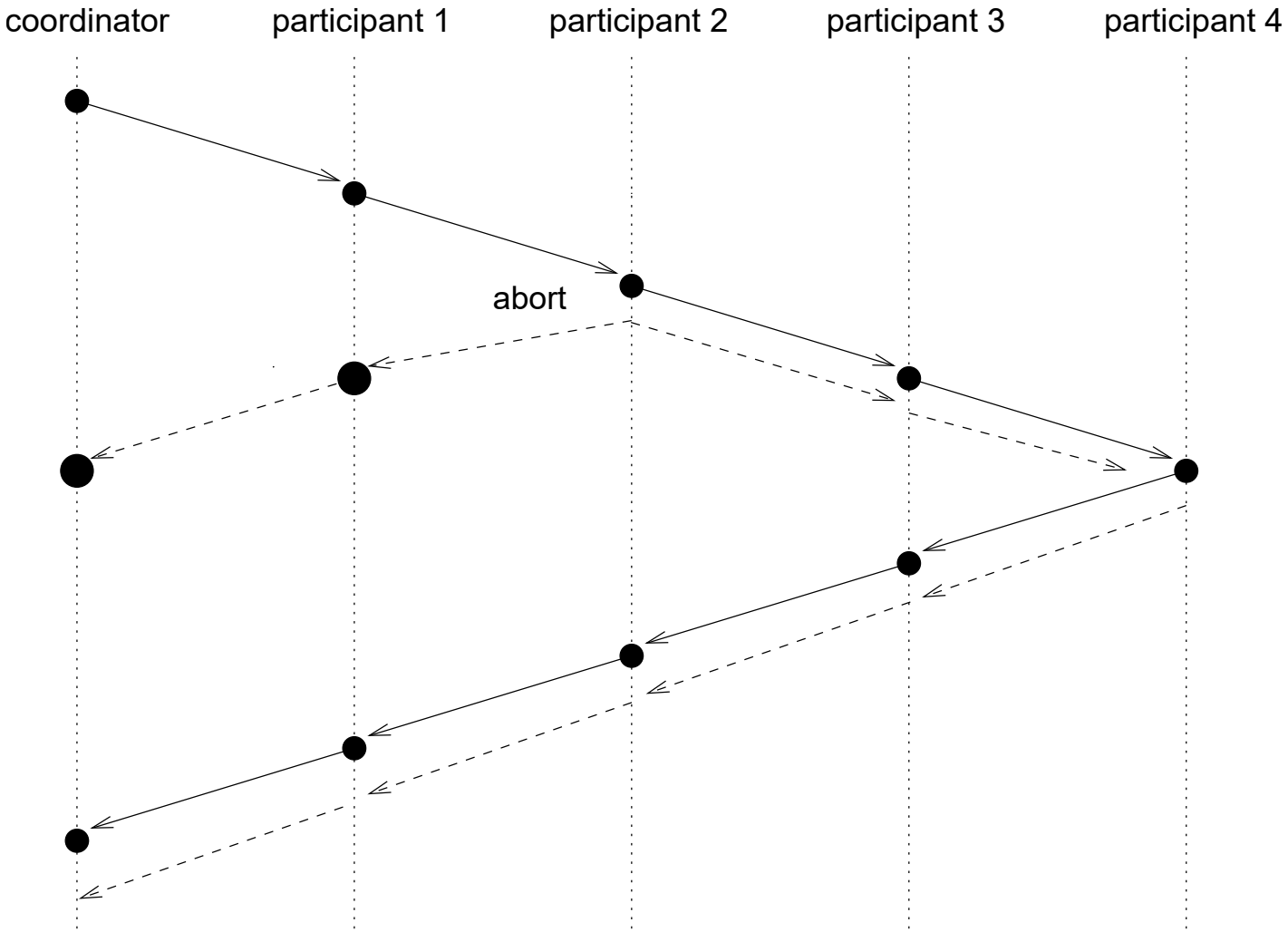
- Participants signaled **`Vote-Commit`** but coordinator fails (1)
  - **`Abort`** of participants after timeout
  - But: Undo of a made decision!
- After sending **`Global-Commit`** (to an unkown number of participants) has been sent, coordinator and participant 1 fail (2)
  - Who sends **`Global-Commit`**? Or **`Abort`**?

# Variants of 2PC I

- Linear 2PC: Coordinator as initiator
  - Coordinator sends **`Prepare-To-Commit`** to participant 1
  - Participant 1 makes a decision and sends it to the next participant
  - **`Vote-Abort`** signal also to predecessor
  - Last participant receives **`Vote-Commit`** and votes “Yes“ $\rightarrow$ **`Global-Commit`** to predecessor

Disadvantage: Slow because of sequential processing

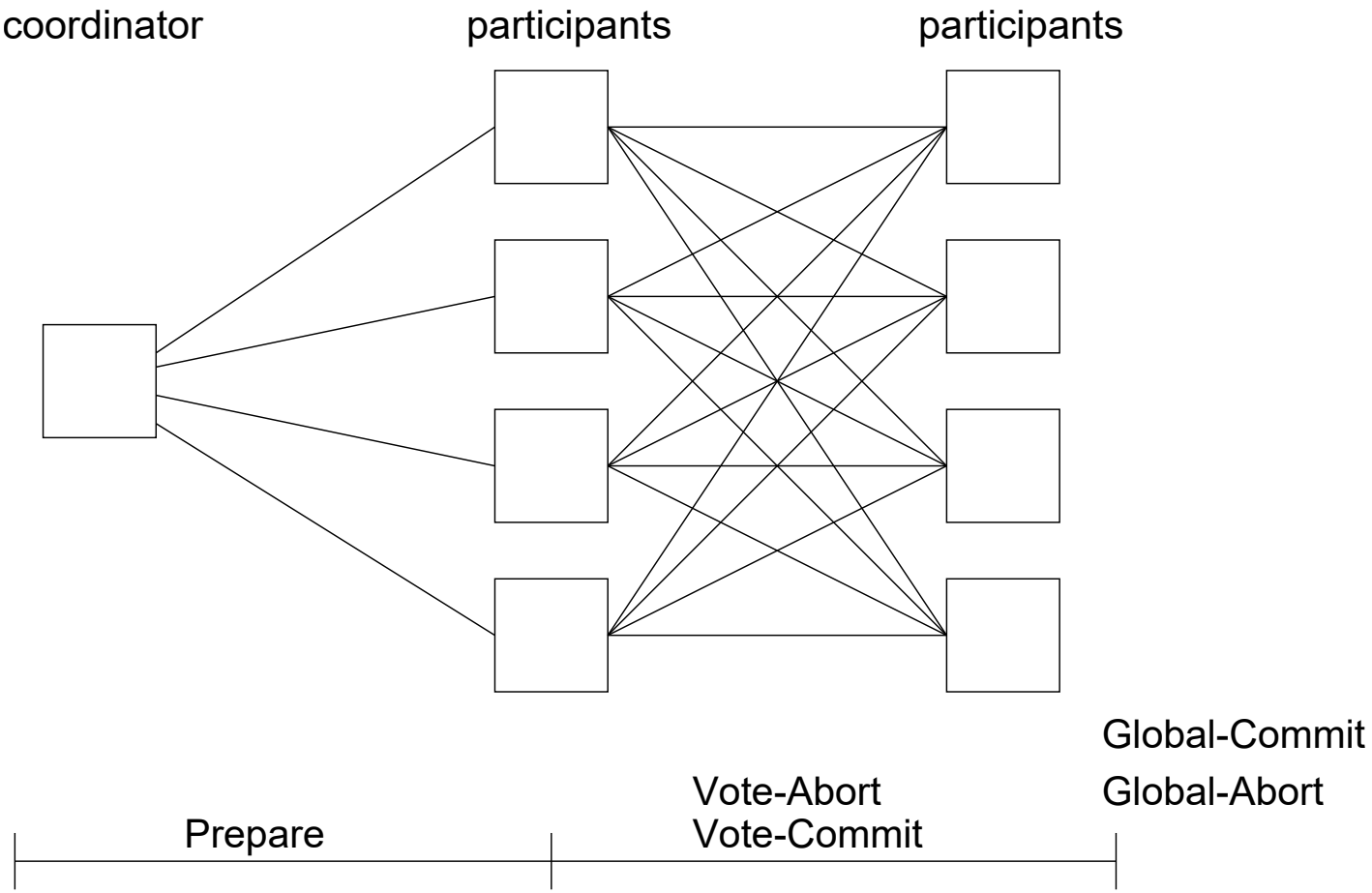
# Linear 2PC: Execution schema



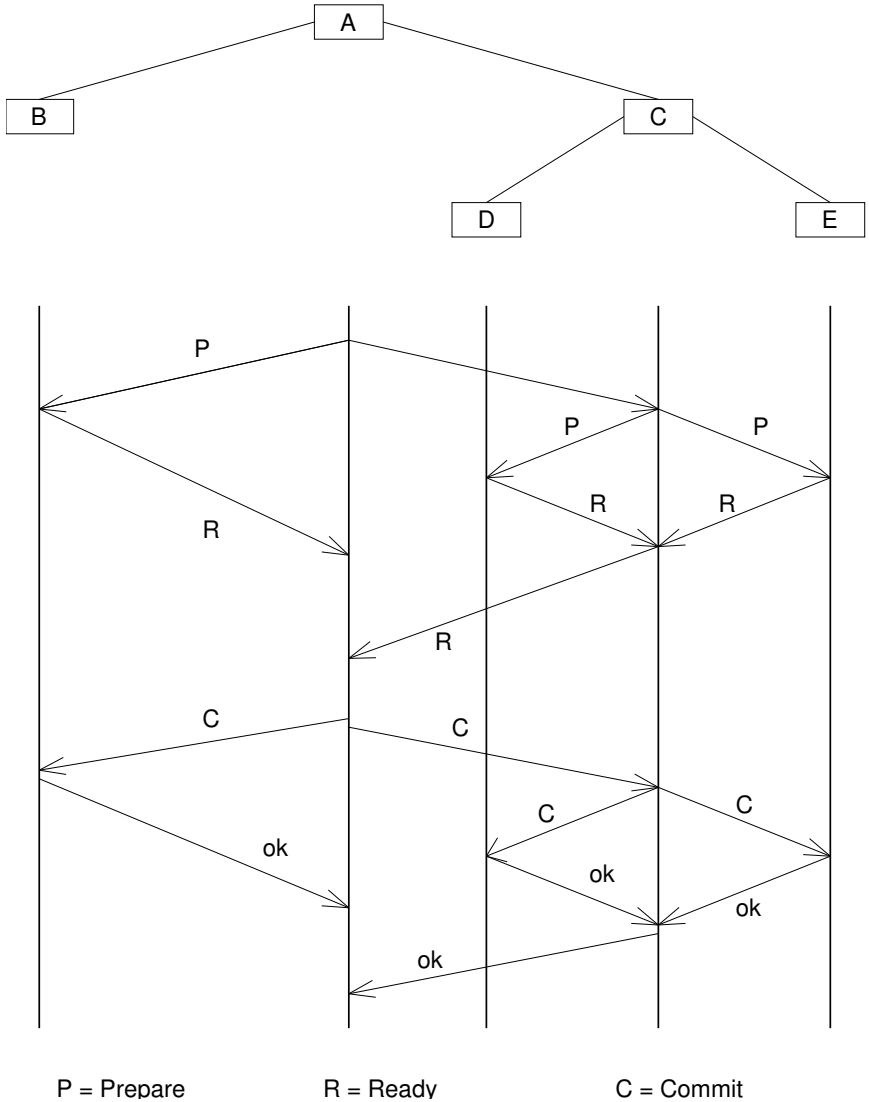
# Variants of 2PC II

- Distributed 2PC: Local voting process
  - Coordinator sends **`Prepare-To-Commit`** to all participants
  - Write decision into Log and forward to all participants
  - Every participant receives all results and makes a local decision
  - Disadvantage: A lot of communication is required
  - Advantage: Quick answers because of missing phase 2
- Hierarchical 2PC: Coordinators and sub-coordinators

# Distributed 2PC: Execution schema



# Hierarchical 2PC: Execution schema



# 3-Phase Commit Protocol

- Problems of 2PC: Failure of coordinators before participants receive **`Global-Commit`** / **`Global-Abort`**
- Solution: 3PC with additional **`PRE-COMMIT`**-phase
  - Participants that receive **`Prepare-To-Commit`** know that **`Commit`** will arrive only if coordinator does not fail
  - Coordinator sends **`Commit`**, only after *k* participants confirm the **`Prepare-To-Commit`** with a **`Ready-To-Commit`**

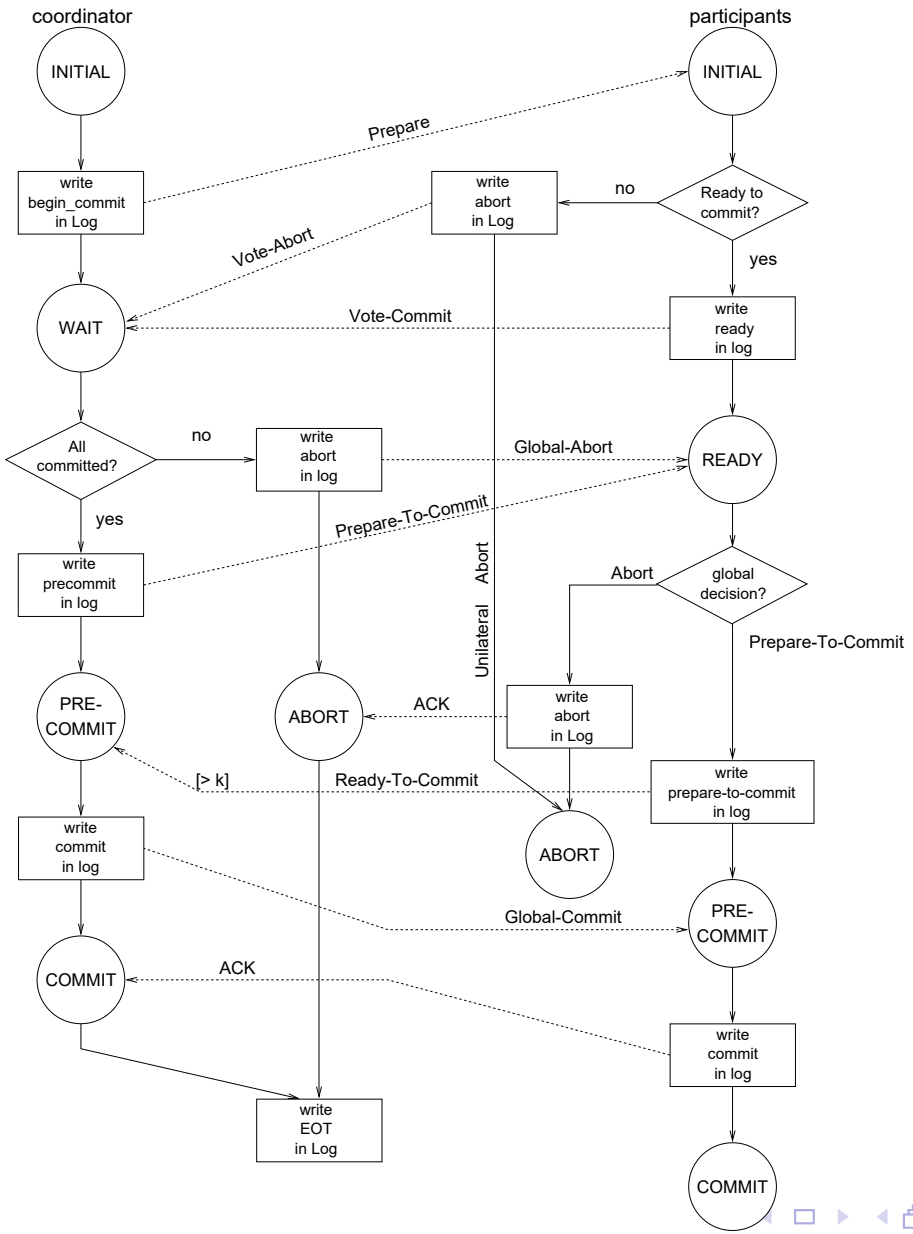
# 3PC: Phases I

1. *Voting phase*
   1. Coordinator sends **`Prepare`** signal
   2. Every participant answers and signals its decision (**`Vote-Commit`** or **`Vote-Abort`**)
   3. In case of **`Vote-Abort`**, directly into state **`ABORT`**
2. *Decision preparation phase*
   1. Coordinator collects decisions; in case of **`Vote-Commit`**, a **`Prepare-To-Commit`** is sent to all; otherwise **`Global-Abort`**
   2. Every participant with **`Vote-Commit`** waits for **`Prepare-To-Commit`** and confirms with **`Ready-To-Commit`**; otherwise **`Global-Abort`**

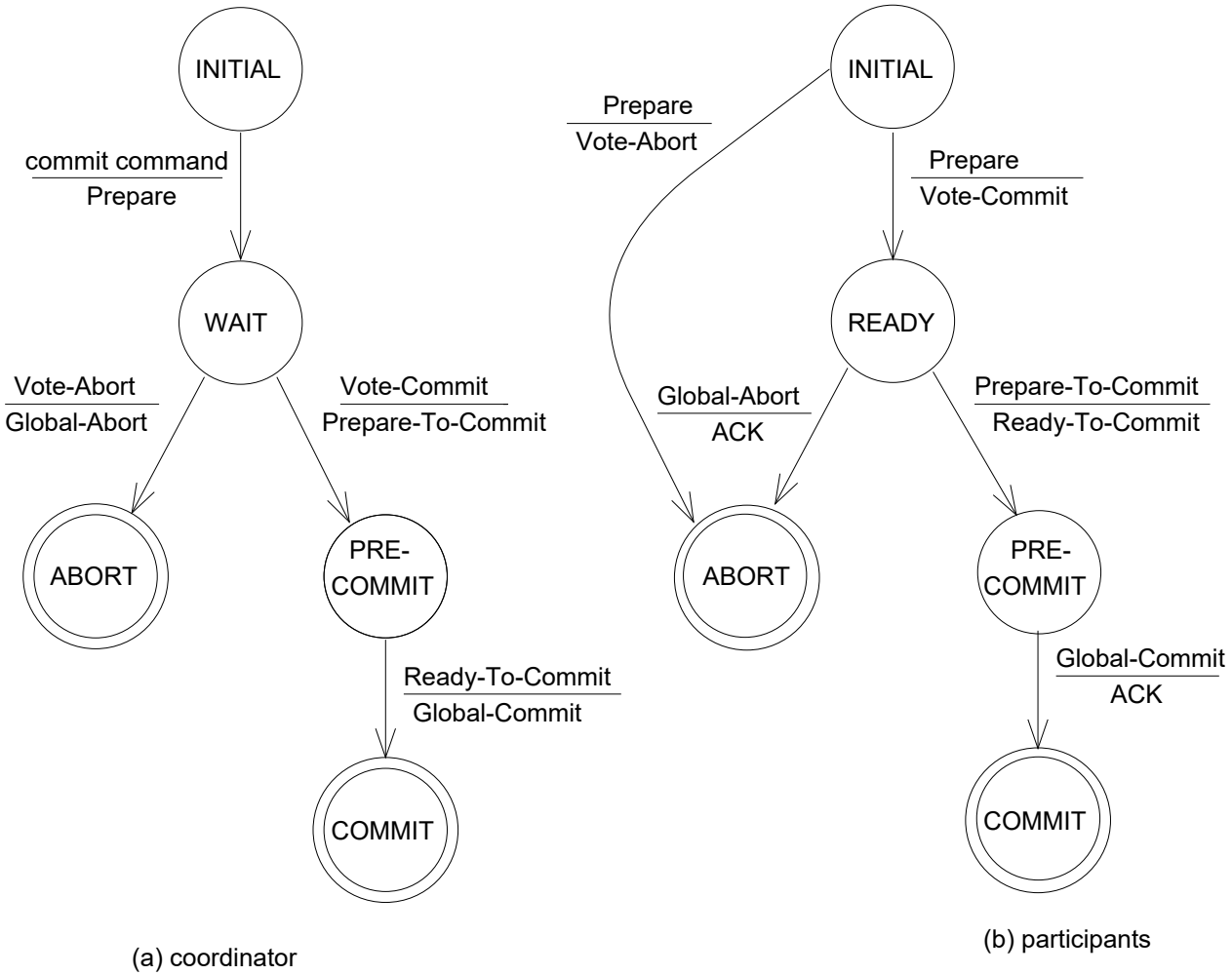
# 3PC Phases II

3. *Decision Phase*
   1. Coordinator collects all confirmations and makes a decision
   2. Participants wait for decision

# 3PC: Execution schema



# 3PC: State Transition



# 3PC: Errors

Failure of coordinator and up to $k - 1$ further participants

1. All further participants in state **`READY`**
   - Failed participants can only be in states **`READY`**, **`ABORT`** or **`PRE-COMMIT`** $\rightarrow$ Abort of transaction
2. One participant in state **`PRE-COMMIT`** or **`COMMIT`**
   - Becomes new coordinator and continues protocol
   - decision was already made for commit

# Distributed Synchronization

- Local synchronization not sufficient
- Example: $T_1$, $T_2$

| Node 1 | | Node 2 | |
|---|---|---|---|
| $T_1$ | $T_2$ | $T_1$ | $T_2$ |
| $r_1(x)$ | | | |
| $w_1(x)$ | | | |
| | | | $r_2(y)$ |
| | | | $w_2(y)$ |
| | $r_2(x)$ | | |
| | $w_2(x)$ | | |
| | | $r_1(y)$ | |
| | | $w_1(y)$ | |

# Distributed Synchronization: II

- Resulting schedule not conflict serializable; however no local synchronization conflict
- Solution:
  - Distributed timestamp-ordering method
    - ★ Total order on timestamps *(Time, Node-ID)*
    - ★ Requires global clock and distributed clock synchronization
  - Central or distributed locking methods
  - ...

# Distributed timestamp-ordering method

Global timestamp *ts* as two-tuple
(local timestamp $ts_l$, *hid* is Host-ID):

$$ts = (ts_l, hid)$$

Order determination:

- Order according to $ts_l$ value
- In case of same $ts_l$ value, decision according to *hid*

# Synchronization of local clocks

- *Usage of global clock:*
  Requires regular synchronization of local clocks $\rightarrow$ often not acceptable
- *Usage of radio-controlled clock*
- *Distributed clock synchronization*
  Synchronization during communication, later time is adopted $\rightarrow$ (unique time)

# Distributed timestamp allocation (via counter)

| Point in time | Node 1 | | Node 2 | |
|---|---|---|---|---|
| | Local TA | timestamps | local TA | timestamps |
| 1 | $T_1$ | 1 | | |
| 2 | | | $T_1$ | 1 |
| 3 | $T_2$ | 2 | | |
| 4 | $T_3$ | 3 | | |
| 5 | | | $T_2$ | 2 |
| 6 | $T_4$ | 4 | | |
| 7 | $T_5$ | 5 | | |
| 8 | | | $T_3$ | 5 |
| 9 | $T_6$ | 6 | | |
| 10 | | | $T_4$ | 6 |
| 11 | $T_7$ | 7 | | |
| 12 | | | $T_5$ | 7 |
| 13 | $T_8$ | 8 | | |

# Transactions on replicas I

A schedule *s* on a replicated database is *1-copy serializable* if there is a serial schedule on a non-replicable database that has the same effect as *s* on a replicable data set.

# Transactions on replicas II

Replication protocol

- *ROWA-Method* (Read One, Write All): Local read and synchronized updates of all replicas
  $\rightarrow$extremely high complexity; some computer nodes might be unavailable
- *ROWAA-Method* (Read One, Write All Available)
- *Voting procedure*: voting procedure or quorum procedure
  - Statistical number of “eligible voters“
  - Dynamical number of “eligible voters“ is depending on environmental influences such as lost connections and access behavior

  (Weighting of votes is possible)

# Transactions on replicas III

Replication protocol

- *Absolutistic approaches:* e.g. primary copy method: A certain node updates a replica in any case. Choice is static or the node has a *token*

# Distributed locking methods

- *Centralized 2PL (C2PL):* Central management of locks on a node
  $\rightarrow$ requires a lot of communication, heavy load for central lock manager (replication protocol needs to be observed)
- *Primary copy 2PL (PC2PL):* Several lock managers on different nodes; each DB-object has exactly one lock manager
  $\rightarrow$ Distribution of lock managing load
- *Distributed 2PL (D2PL):* Lock manager on every DBMS; lock manager is responsible for its own DB-objects
  (no replication $\rightarrow$ PC2PL, otherwise ROWA)

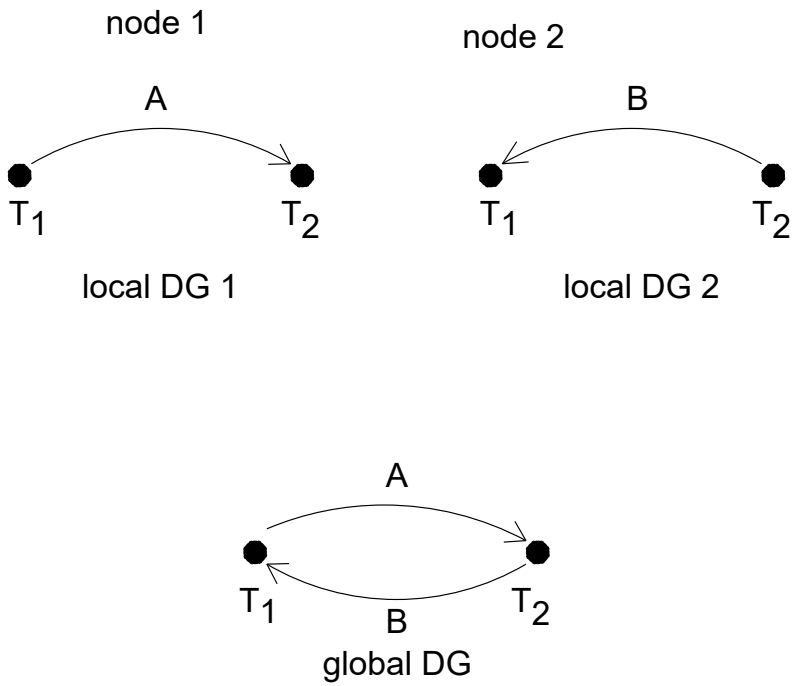
# Distributed Deadlocks

Classes of deadlock handling:

- *Deadlock-free*
  Preclaiming (C2PL) – atomic requirement can hardly be fulfilled in distributed cases
- *Deadlock prevention*
  Total order of objects and their occupancy
- *Deadlock detection*
  Detection of distributed deadlocks is problematic

# Deadlock detection

- *Time-Out-Mechanism*
- *Global deadlock graph*
  Central coordinator manages conflict graph (coordinator could fail!)



# Deadlock handling

Practical methods

- *Conservative locking*
  C2PL-method: Problems with atomicity
- *Timestamps as requirement order*
  Timestamps for handling lock conflicts
- *Deadlock detection*
  - *Centralized:* Central vertex manages complete wait graph
  - *Hierarchical deadlock detection:* Many deadlocks can be identified locally; difficult implementation though
  - *Distributed deadlock detection*

# Distributed detection of global deadlocks I

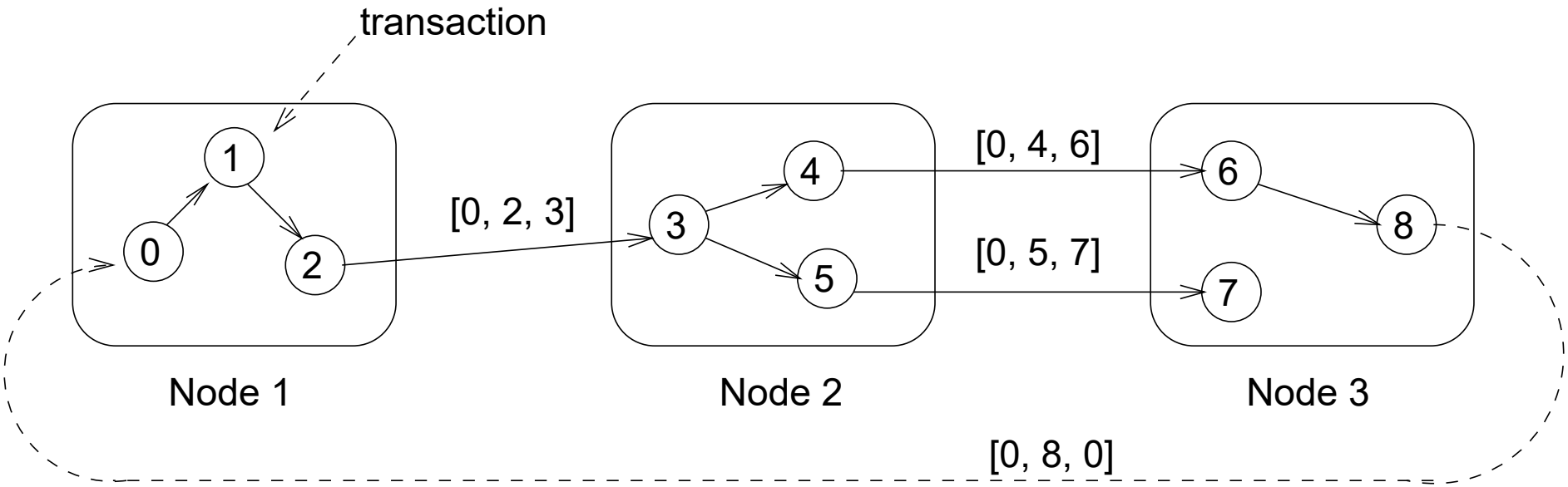
Deadlocks do not exist locally in any computer node. Computers send each other messages of the form $[m, n, k]$.

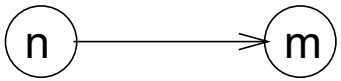
1. $m \cong$ Number of blocked process
2. $n \cong$ Number of transaction that sent the message (sender)
3. $k \cong$ Number of transaction to which the message is directed (receiver)

# Distributed identification of global deadlocks: II

- Message transmission starts with $[0, 0, 1]$
- Message $[0, 2, 3]$ means that the blocked transaction is transaction 0, sender is transaction 2 and receiver is transaction 3
- A deadlock occurs if and only if the message arrives at the blocked process

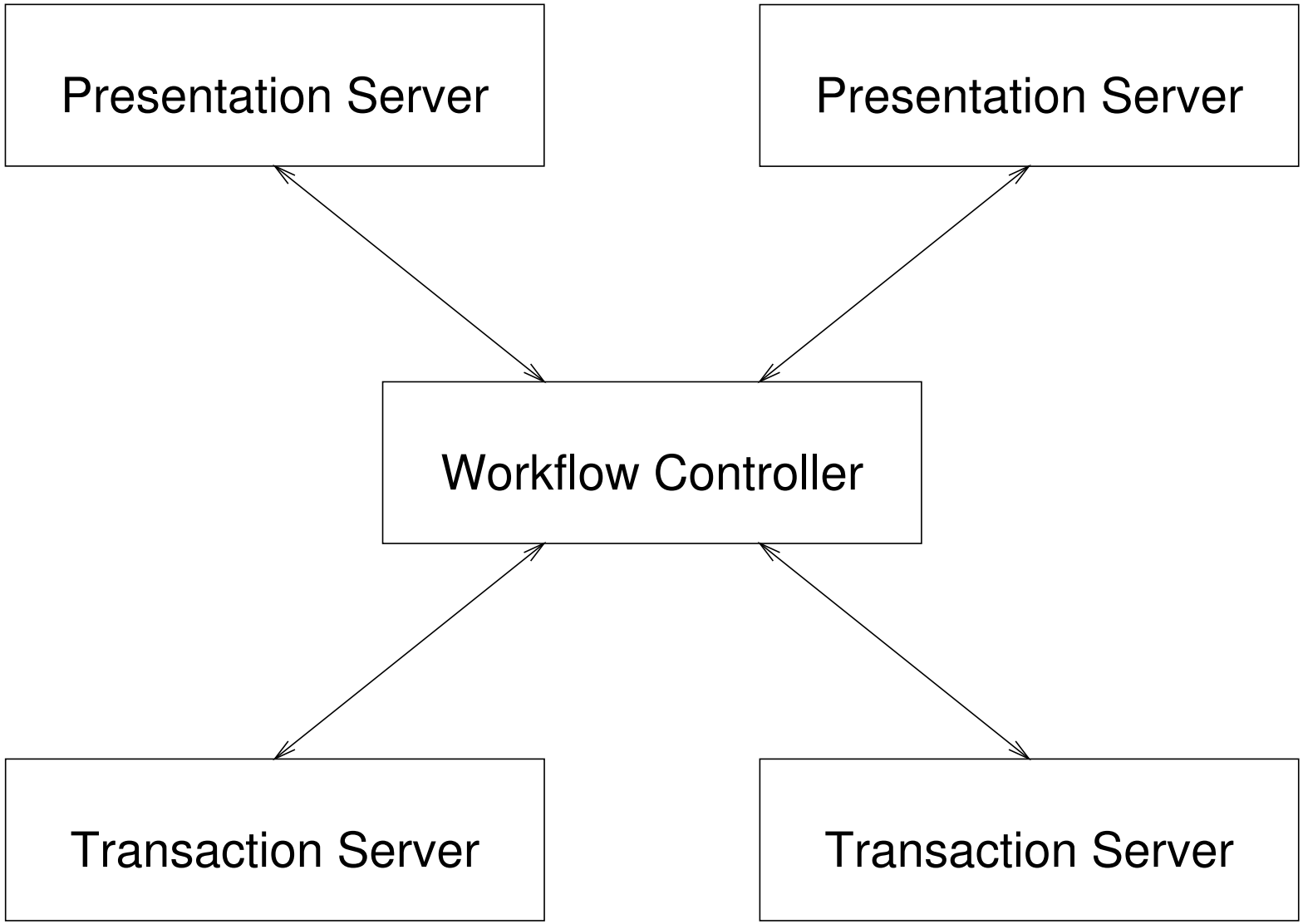
# Distributed identification of global deadlocks: Example




The transaction m waits for transaction n or
the transaction m is blocked by transaction n

# Transaction monitors



# Transaction monitors: Architecture

- Presentation server, acts as client and realizes communication with user (command language or menu-driven interfaces for sending transactions etc.)
- Workflow controller forces routing of transaction requirements of different DBMS and realizes, for instance, two-phase commit protocol
- Transaction server realizes connection of local DBMS with transaction monitor

# Advantages of a transaction monitor

- Offers *one* standardized interface for programming transactions on different DBMS
- In distributed processing, it manages routing of transactions and forces commit protocols
- Offers systems functions such as load balancing, error control, and system configuration
- Is able to fulfill functions such as writing log files or monitoring communication
- Transaction server of a TP-monitor can also encapsulate data that is not managed by a DBMS with full transaction functionality