Databases

http://www.dcs.fmph.uniba.sk/~plachetk

/TEACHING/DB2

http://www.dcs.fmph.uniba.sk/~sturc/databazy/rldb

Tomáš Plachetka, Ján Šturc

Faculty of mathematics, physics and informatics,
Comenius University, Bratislava

Summer 2023–2024

Literature

P.A. Bernstein, V. Hadzilacos, N. Goodman: Concurrency Control and Recovery in Database Systems, 1987

H. Garcia-Molina, J.D. Ullman, J. Widom: Database System Implementation, Prentice Hall, 2000

P.A. Bernstein, E.Newcomer: Transaction Processing, 2009

G. Weikum, G. Vossen: Transactional Information Systems, 2002

General requirements: ACID

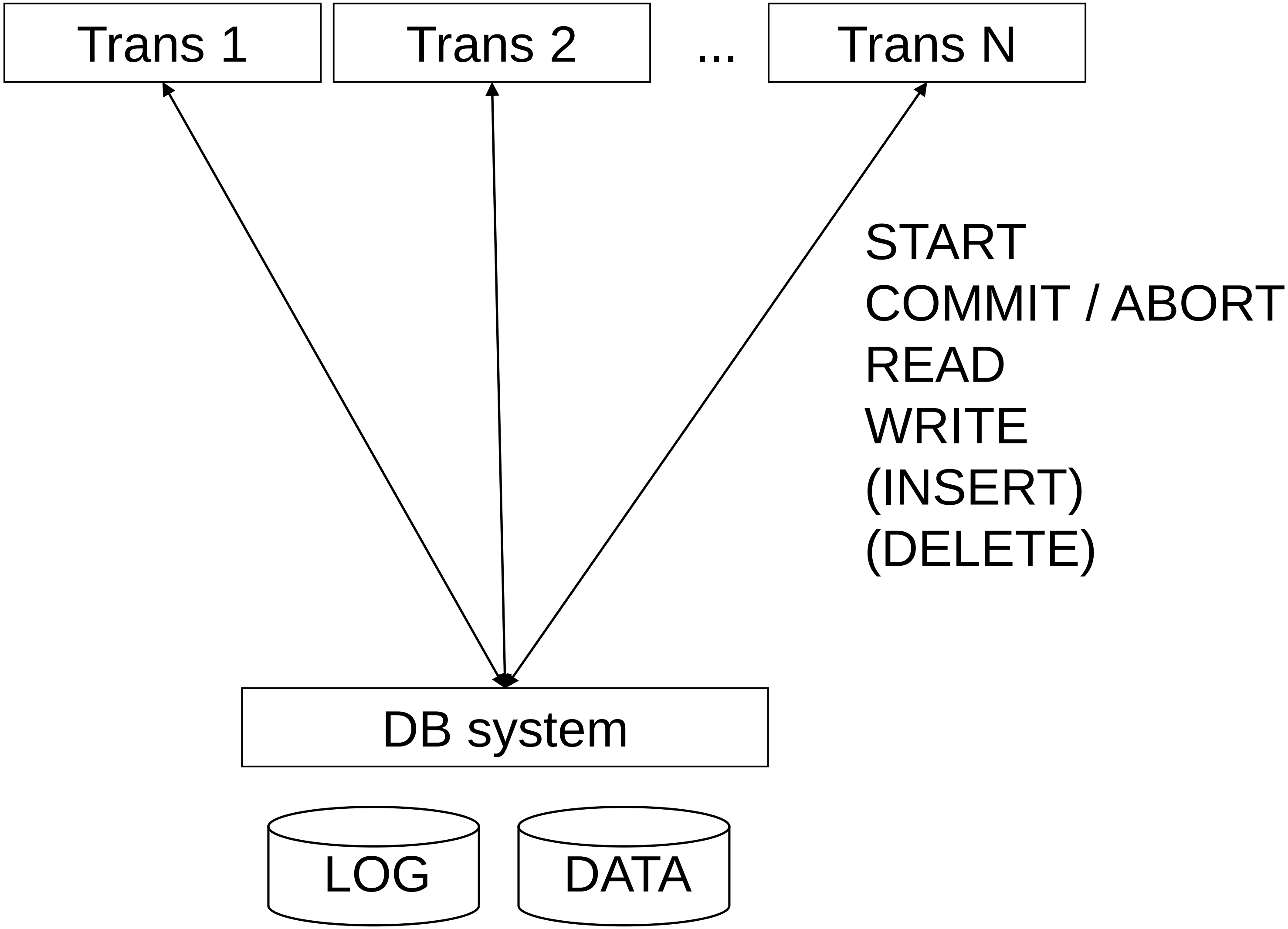
Atomicity: each transaction is processed in whole (if any part fails, then the whole transaction fails and the database is left unchanged)

Consistency: each single transaction which begins with a consistent database leaves the database in a consistent state when it is commited (**this requirement is addressed to application programmers**)

Isolation: the transactions are either executed serially, or the system guarantees that the execution has the same effect as a serial execution

Durability: once a transaction is committed, all the changes made by the transactions remain in the database, also in presence of failures

Centralised (2-tier) architecture



START
COMMIT / ABORT
READ
WRITE
(INSERT)
(DELETE)

Centralised (2-tier) architecture

Advantages:

- simple
- implemented and tested over decades
- compatible with existing bureaucracy

Disadvantages:

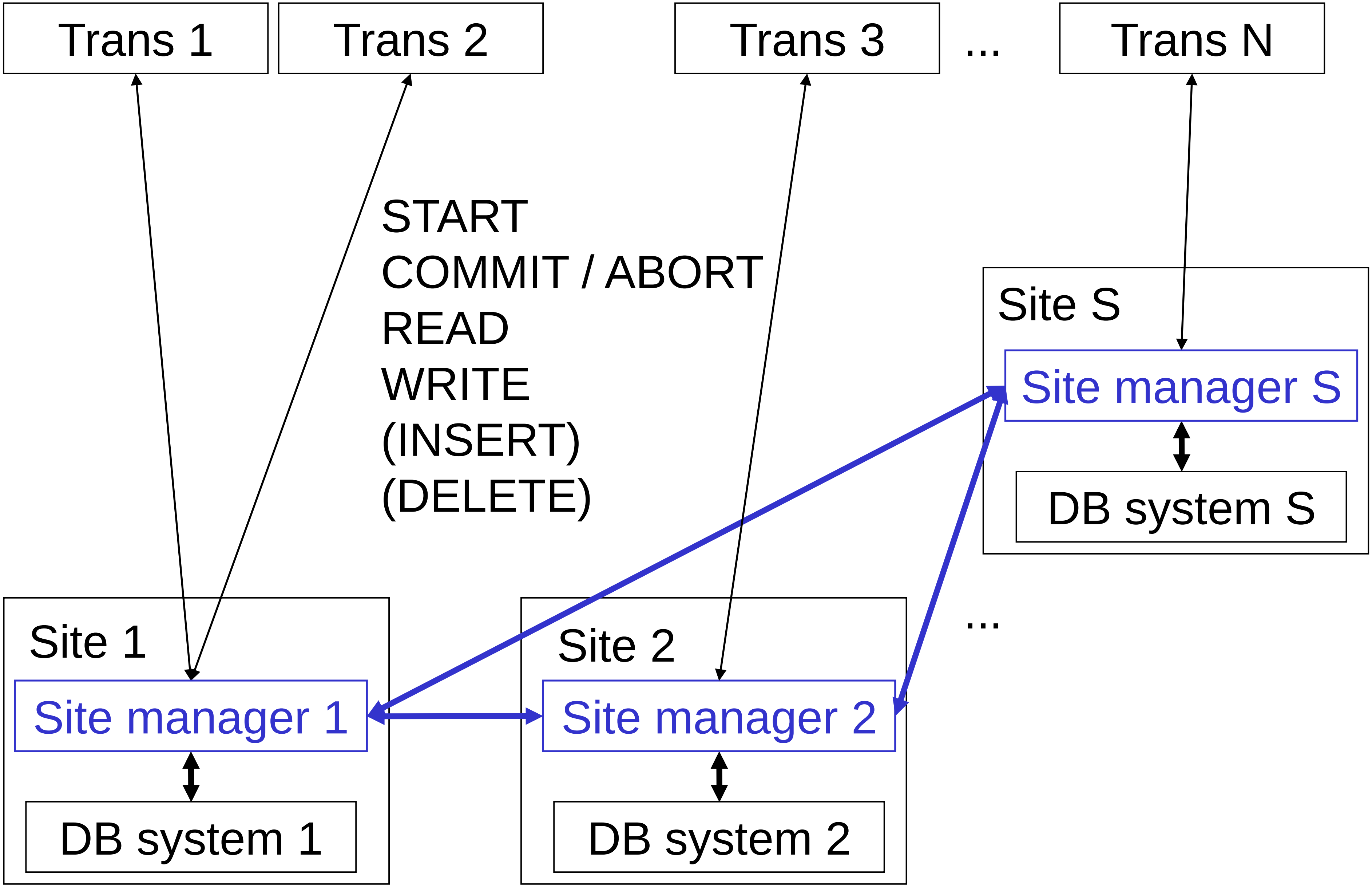
- everything else
- fragility (data cannot be accessed when the server is out of order)
- no scalability
- etc.

Distributed databases: a framework

Requirements:

- ACID
- Transparency: the client does not see whether the system is distributed or not
- Resistance against failures: a server must not wait indefinitely long for a recovery of another server

Distributed (3-tier) architecture



Trans 1

Trans 2

Trans 3

...

Trans N

START
COMMIT / ABORT
READ
WRITE
(INSERT)
(DELETE)

Site S

Site manager S

DB system S

Site 1

Site manager 1

DB system 1

Site 2

Site manager 2

DB system 2

...

Atomic commit protocol

A transaction demands COMMIT from its home-site. (Note that even if the transaction could see all sites, it cannot demand COMMIT from all the sites at once.)

The most important requirement: Either all sites agree on COMMIT, or all agree on ABORT

Assumptions:

- If a site fails, it simply stops working (no byzantine failures). However, a site can fail *during* the agreement protocol
- A failure of a site is detected (when a site is waiting for a message from a failed site, it receives a message "I am down" from the network on behalf of the failed site)

Atomic commit protocol: requirements

1. All sites which decide will make the same decision (all decide COMMIT or all decide ABORT)

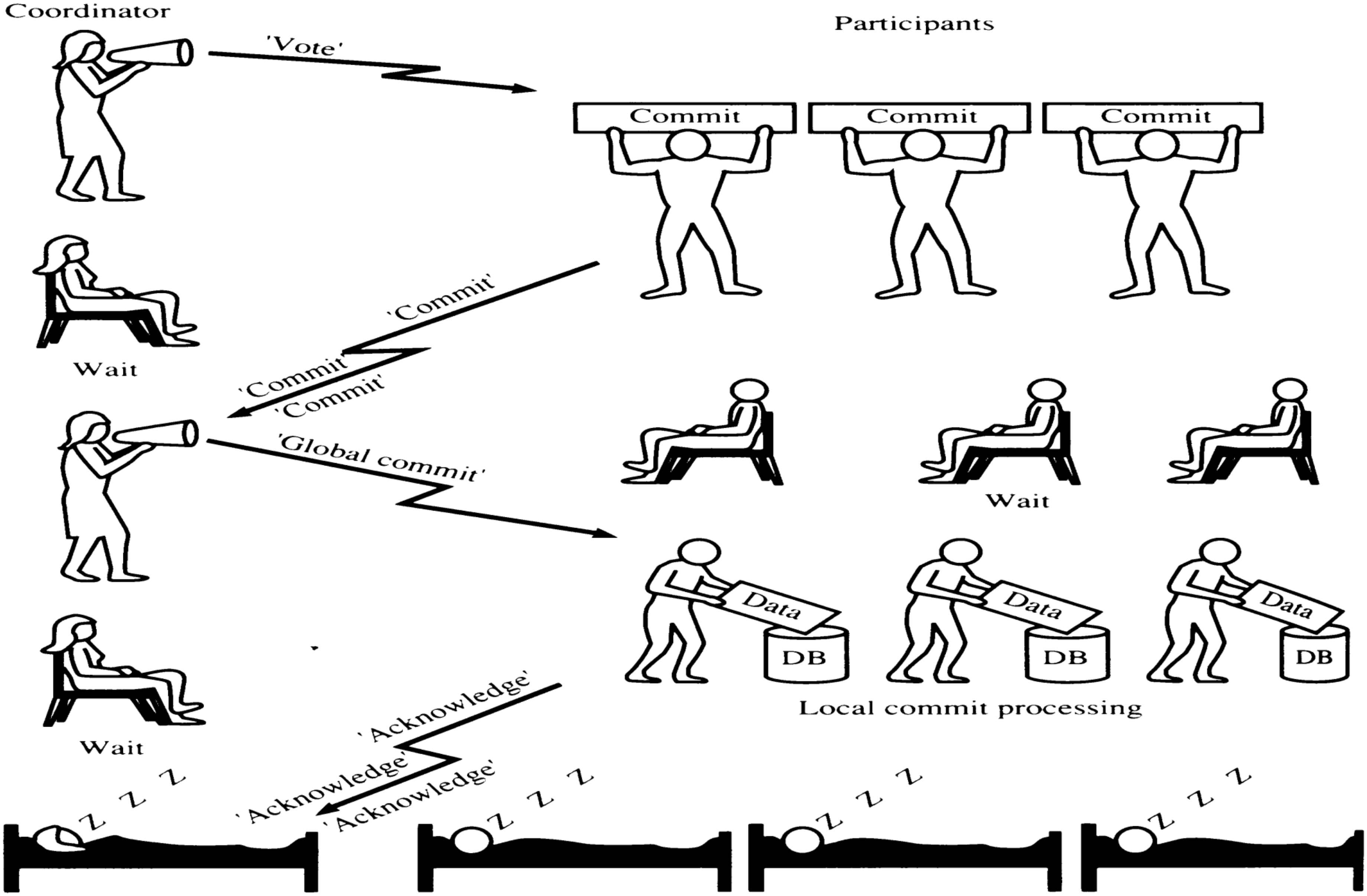
2. If a site decides, it never changes its decision

3. The decision COMMIT can be only made when all the sites vote YES

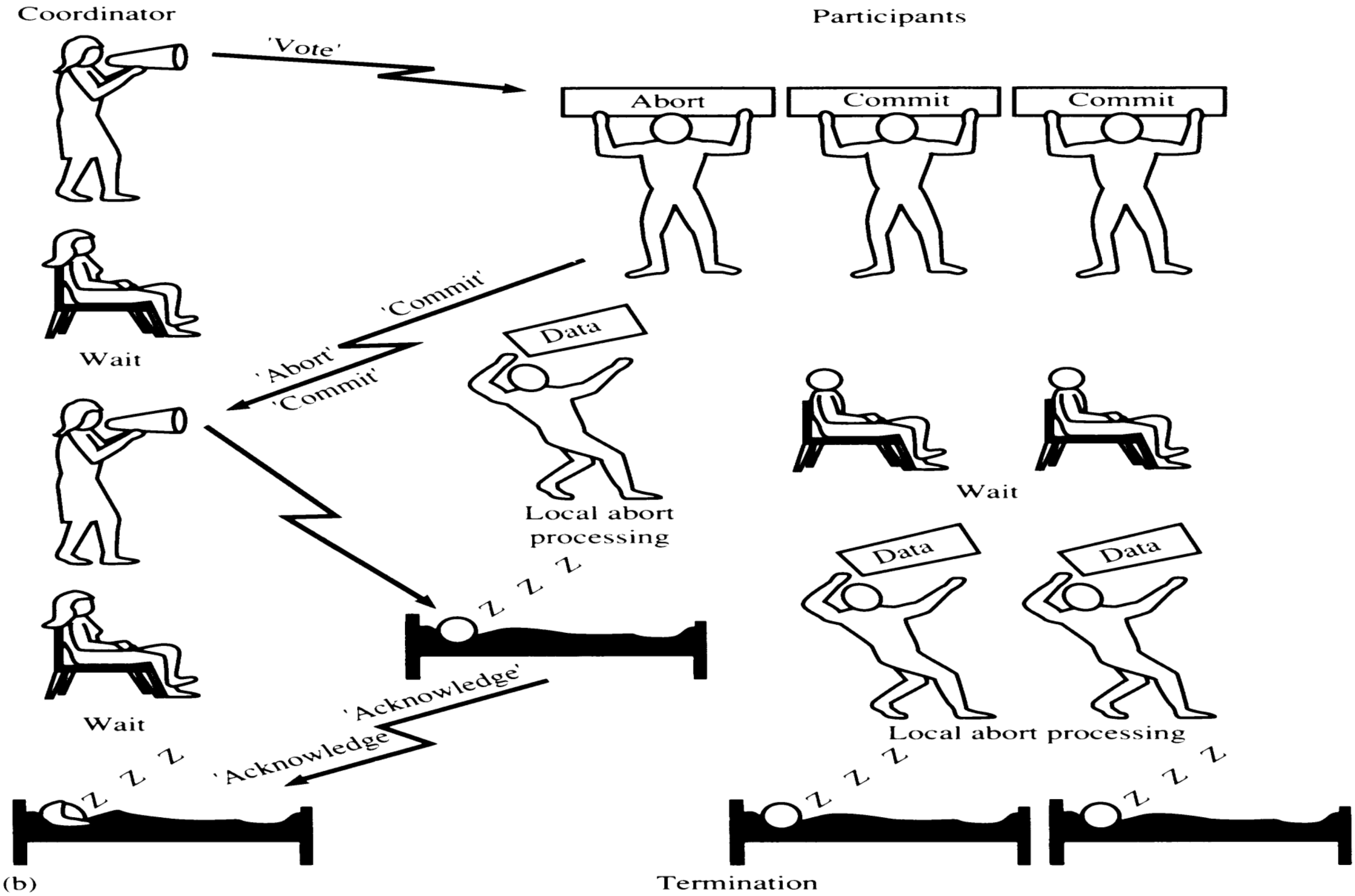
4. If all the sites vote YES and there are no failures, then all the sites must eventually decide COMMIT

5. If there are no failures for a sufficiently long time, then the non-failed sites will eventually make a decision

Two-phase atomic commit protocol (2ACP): COMMIT



Two-phase atomic commit protocol (2ACP): ABORT



Two-phase atomic commit protocol (2ACP): Phase 1

Phase 1: a transaction T demands COMMIT from the coordinator. If the coordinator votes NO, it immediately decides ABORT (writes <ABORT T> to its log) and broadcasts [ABORT] to all the participants. Otherwise:

- Coordinator writes <prepare T> to its log and broadcasts [VOTE T] to all the participants
- When a participant receives [VOTE T], it asks its underlying database system whether it can commit T
  - –If not, it decides ABORT (it writes <ABORT T> to its log) and answers the coordinator with [NO T]
  - –If yes, it writes <YES T> do its log and answers the coordinator with [YES T]

Two-phase atomic commit protocol (2ACP): Phase 2

Phase 2: The coordinator is waiting for YES/NO from all the participants

- When it receives a [NO T], it decides ABORT, writes <ABORT T> to its log and broadcasts [ABORT T]
- When all the answers are [YES T], it decides COMMIT, writes <COMMIT T> do its log and broadcasts [COMMIT T]

The participants (which voted YES) are waiting for the decision from the coordinator and then write the decision to their logs

When a participant fails before sending its vote, the coordinator detects the failure and proceeds as if the participant voted NO

A failed participant does not influence the decision

2ACP: Recovery from a failure

A participant recovering from a failure reads its log.

When it finds <COMMIT T>, it does redo(T)

When it finds <ABORT T> or <NO T>, it does undo(T)

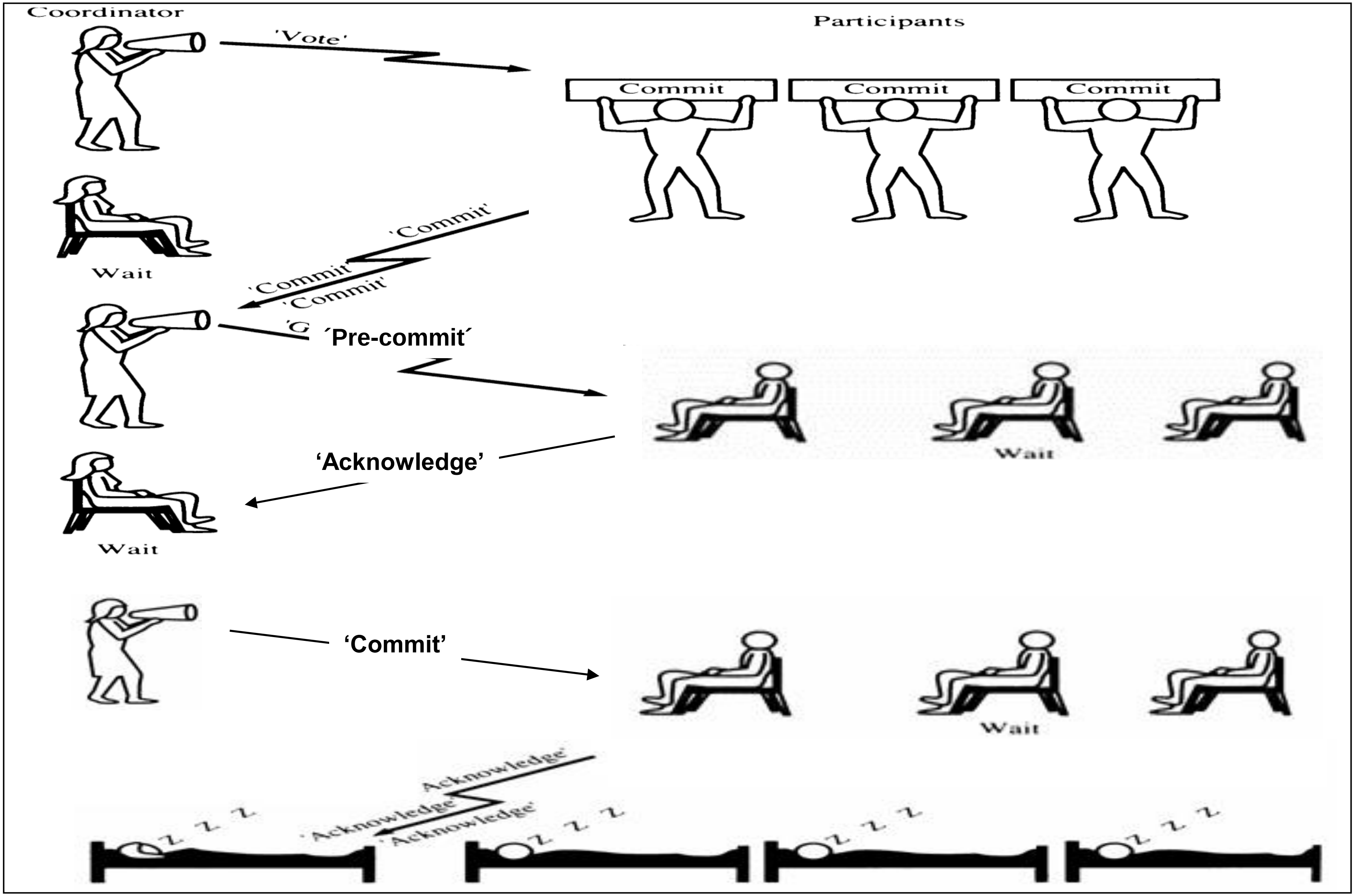
When it finds [YES T], it notifies the coordinator and waits for the decision (subsequently, it does undo(T) or redo(T))

The coordinator recovering from a failure reads its log.

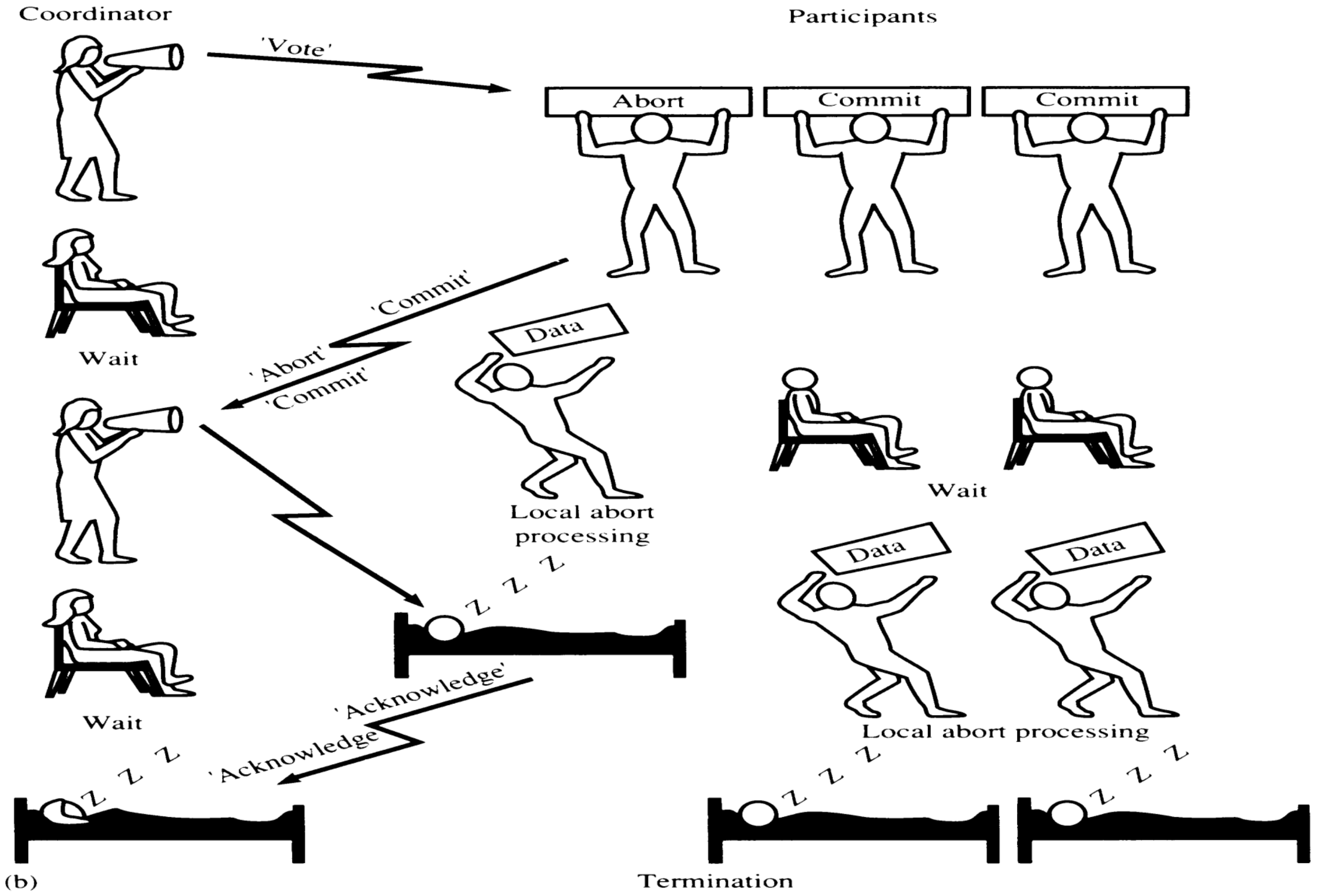
When it finds <COMMIT T>, it does redo(T) and broadcasts <COMMIT T> to participants

Otherwise it does undo(T) and broadcasts <ABORT T> to participants

Three-phase atomic commit (3ACP): COMMIT



Three-phase atomic commit (3ACP): ABORT



3ACP: the idea behind the PRE-COMMIT phase

The idea behind the middle phase (PRE-COMMIT) is to alleviate a possible failure of the coordinator. When the coordinator fails, the surviving sites have a piece of information which allows for a decision: either a non-failed participant has already a received a PRE-COMMIT message or not:

- If not, the decision will eventually be ABORT.
- If so, the decision will be COMMIT, if there are no subsequent failures

After the failure of the coordinator, the participants elect a new coordinator which collects the information on the presence of PRE-COMMIT. When it finds one, it first floods all the non-failed processes with PRE-COMMIT, waits for acknowledgements and then decides COMMIT (this decision is safe, because even when it fails, the others will COMMIT anyway, using the same scenario)

3ACP: link failures

The aforementioned version of 3ACP assumes that only the sites may fail, not the links which connect them (partitioning of the network)

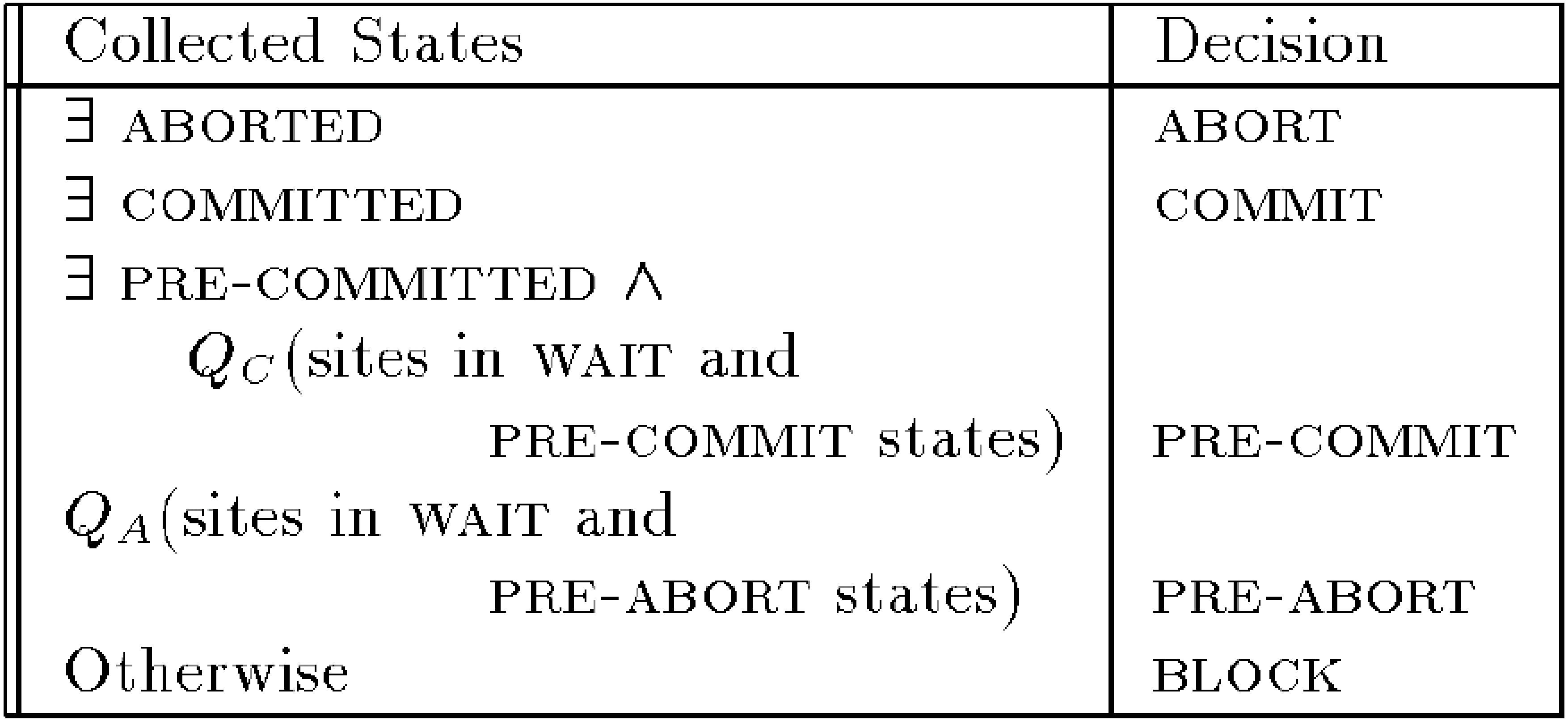
When the links are allowed to fail as well, 3ACP can be extended with a majority rule [Skeen, 1982] and PRE-ABORT states [Keidar, Dolev, 1994]. This extended 3ACP guarantees correctness, but blocks sometimes. More precisely, it blocks exactly when it has to (i.e. when any other atomic commit protocol would also block)

If link failures or total site failures (all sites fail) are possible, then every atomic commit protocol must block sometimes

If all sites fail, all recovered sites must wait until the last failed site has recovered (or some site has reached a decision before)

3ACP: link failures

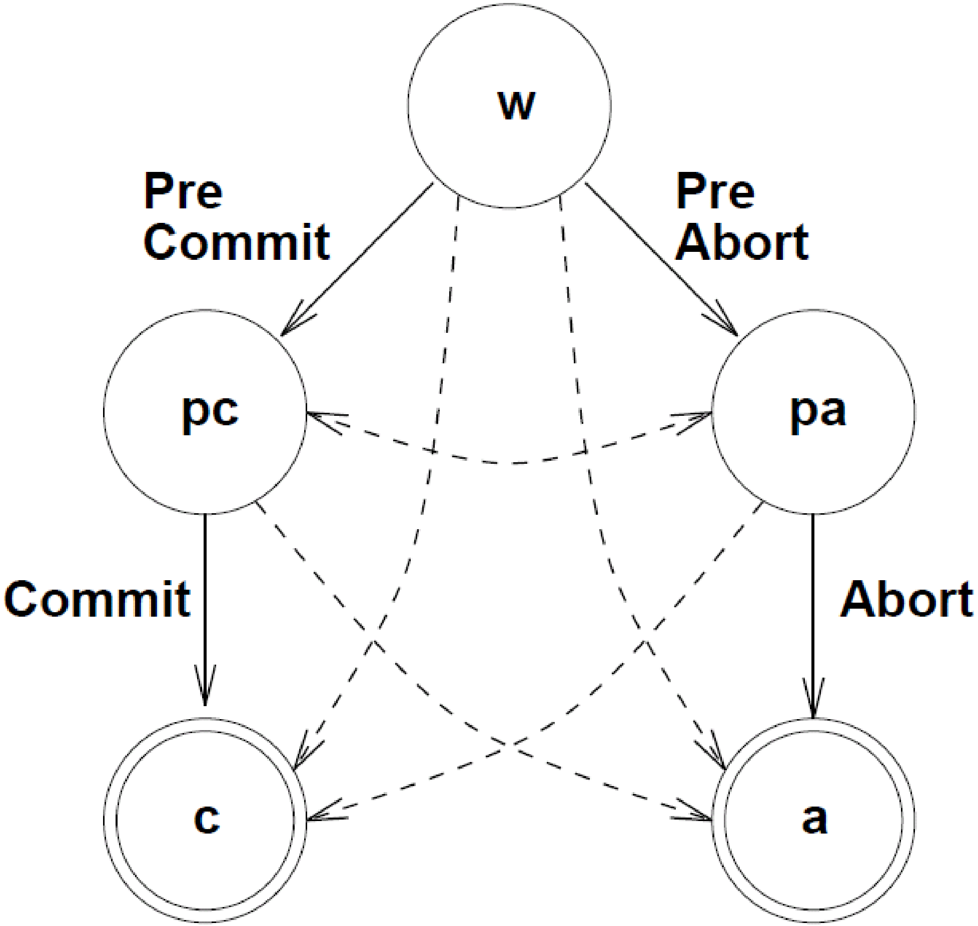
Majority protocol: a group of sites is allowed to make a decision only when it constitutes a strict majority of all sites



Collected States	Decision
∃ ABORTED	ABORT
∃ COMMITTED	COMMIT
∃ PRE-COMMITTED $\wedge$ Q_C(sites in WAIT and PRE-COMMIT states)	PRE-COMMIT
Q_A(sites in WAIT and PRE-ABORT states)	PRE-ABORT
Otherwise	BLOCK

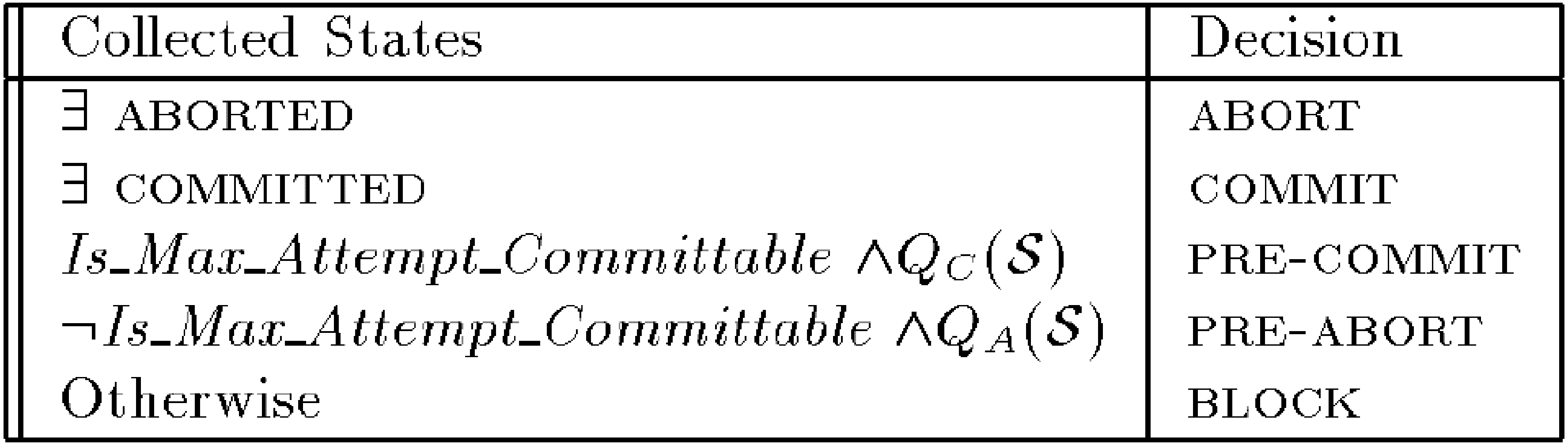
3ACP: link failures

Majority protocol must be extended with PRE-ABORT states. Another problem arises in the majority extended with PRE-ABORT states: connected nodes form a quorum but the number of PRE-COMMITs equals the number of PRE-ABORTs



3ACP: link failures

Keidar&Dolev, 1994: a tie break for breaking the symmetry are two counters in each site which encode whether the site was last involved in a commit or abort attempt



Collected States	Decision
$\exists$ ABORTED	ABORT
$\exists$ COMMITTED	COMMIT
$Is\_Max\_Attempt\_Committable \wedge Q_C(\mathcal{S})$	PRE-COMMIT
$\neg Is\_Max\_Attempt\_Committable \wedge Q_A(\mathcal{S})$	PRE-ABORT
Otherwise	BLOCK

Election of new coordinator

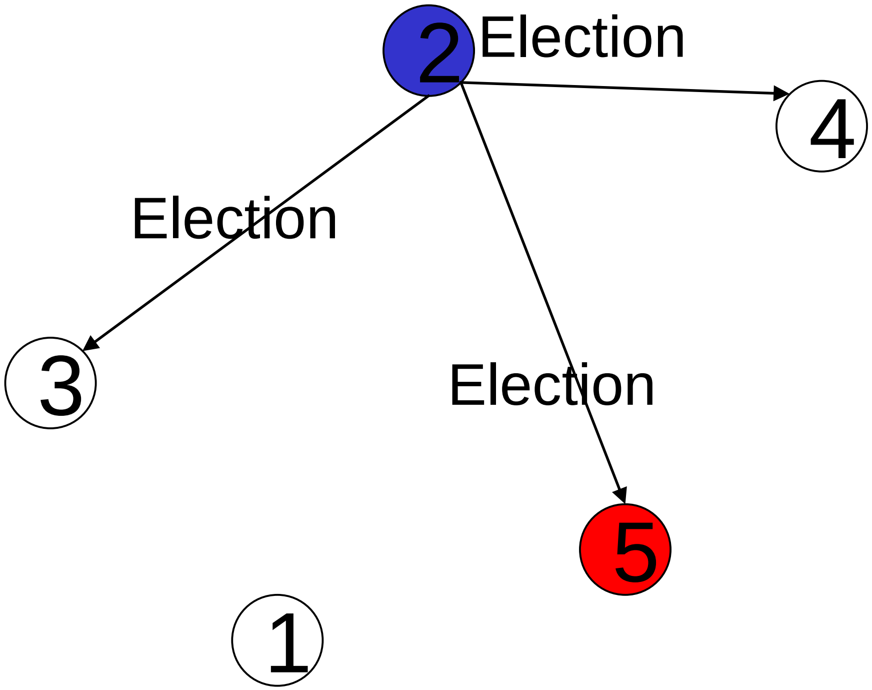
Problem statement:

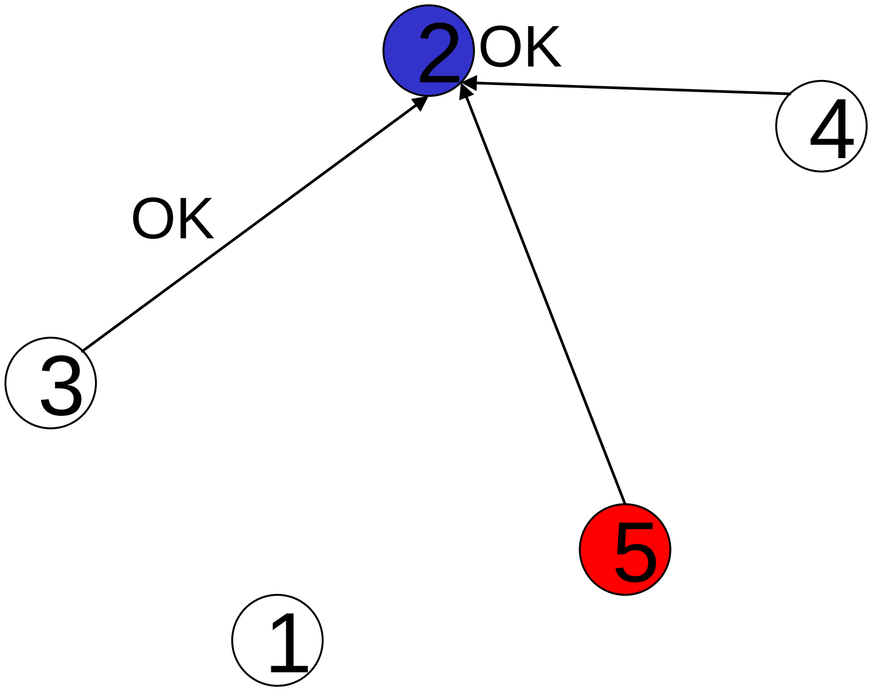
- Each site has a unique identifier
- Each sites knows all identifiers

The goal is to choose the **unfailed** node with the largest identifier to be the coordinator and let all the sites know its identifier

Election of new coordinator: bully protocol

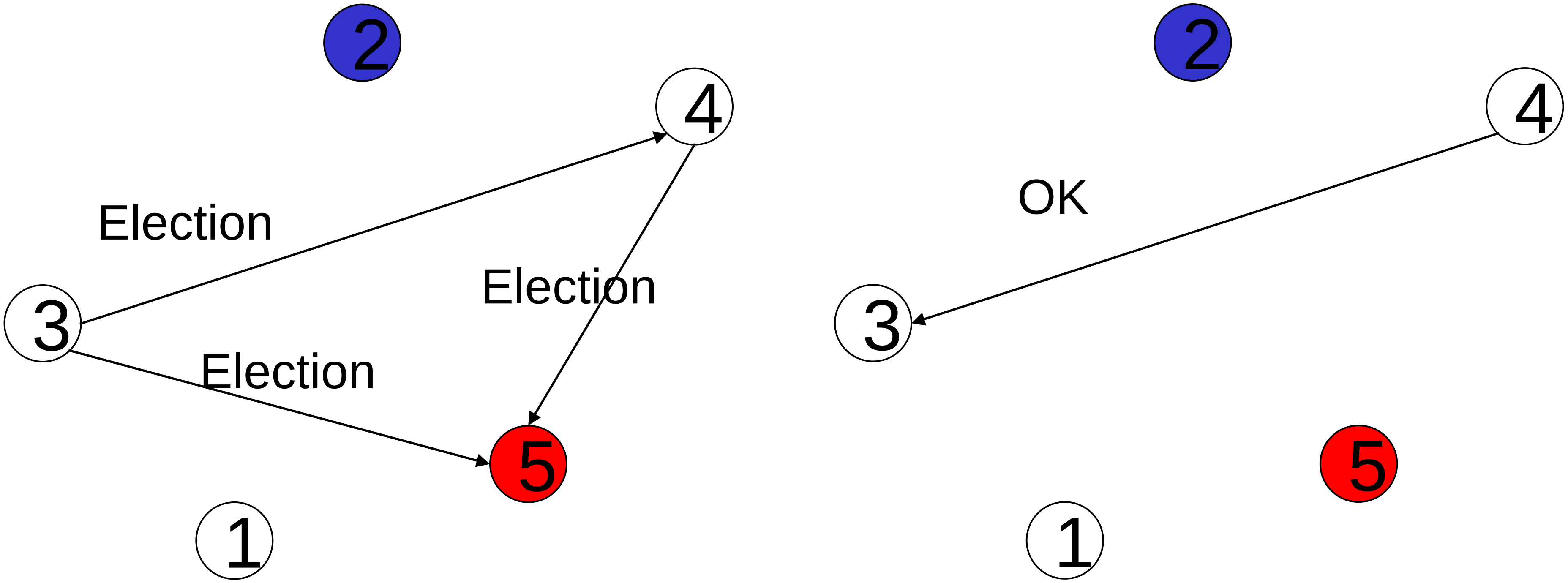
Example: the node 5 has failed





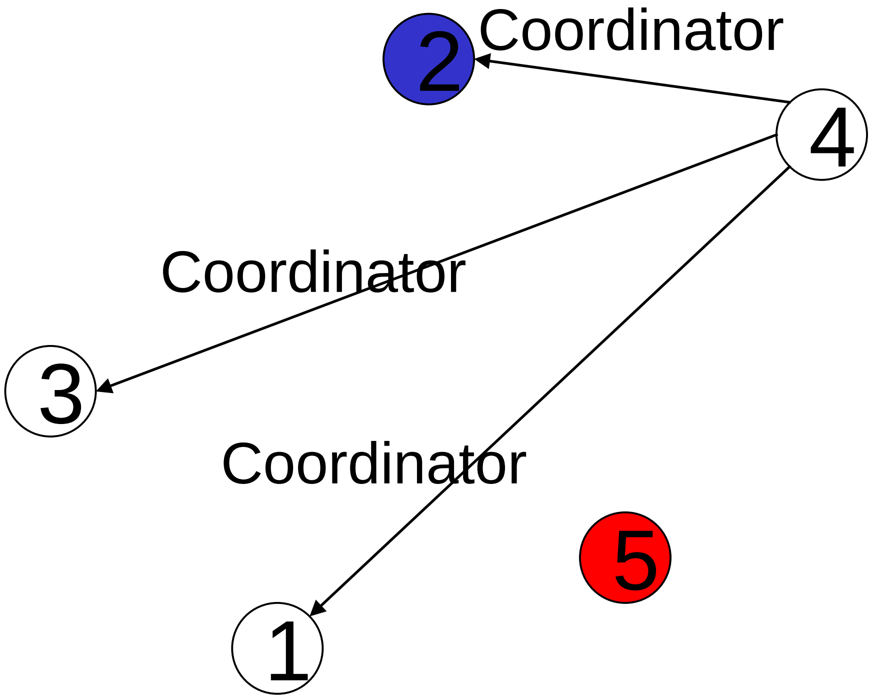
Election of new coordinator: bully protocol

Example: the node 5 has failed



Election of new coordinator: bully protocol

Example: the node 5 has failed, 4 becomes the new coordinator



Replication

Problem statement:

- A data object is replicated in several sites

Goal: Guarantee the isolation of transactions → distributed locking, distributed time-stamp protocol, ...

Distributed 2-phase locking: centralised

Centralised scheme

- One site is a manager for all locks
  - Easy to implement
  - The manager can fail; the manager is a bottleneck

Distributed 2-phase locking: primary copy

Primary-copy scheme

- Each site is a lock manager, responsible for a subset of the data objects
- In case of replicated objects, one copy is the primary one (i.e. **exactly one site manages the primary copy**)
- Read-lock can be acquired from any site which has the replica of the object
- Write-lock can only be acquired from the primary-copy manager, which must in turn acquire the write-lock from all other sites holding a replica of the object
  - No bottleneck for reading
  - Primary copy manager can fail

⟶ Primary-copy scheme is as bad as the centralised scheme

Distributed 2-phase locking: 2 extremes

Distributed 2-phase ROWA (Read-One-Write-All) protocol

- Read-lock can be acquired from any site manager
- Write-lock must be acquired from all the sites holding a replica of the data object
- Cheap reads, expensive writes

Majority 2-phase protocol

- Read-lock as well as write-lock must be acquired from the strict majority of sites which hold replicas
- 2 (n/2 + 1) messages for lock, (n/2 + 1) messages for unlock
- The price of a read equals the price for a write

(Note that a deadlock may occur in the majority protocol also when acquiring write-lock for a single object:

e.g. there are 3 transactions, each of them has acquired the write-lock in 1/3 sites.)

Distributed 2-phase locking: quorum protocol

Quorum protocol is a compromise between ROWA and the majority protocol

- Each site is assigned a weight w_i, $w_i>0$
- Let S denote the sum of all weights: $S=\sum w_{i,}$
- Let Q_r (read quorum) a Q_w (write quorum) positive numbers such that $\mathbf{Q_r + Q_w > S}$ and $\mathbf{2\ Q_w > S}$
(Q$_r$ a Q$_w$ may even differ for different data objects)

Rules:

- A read-lock must be acquired in sites with the total sum of weights at least Q_r
- A write lock must be acquired in sites with the total sum of weights at least Q_w

Distributed deadlocks

- Deadlocks are more complicated in a distributed system than in a centralized one

Example: T1 runs in Site 1, T2 runs in Site 2

Site 1 (manages X)	**Site 2 (manages Y)**
write-lock1(X)	
w1(X)	
	write-lock2(Y)
	w2(Y)
	write-lock2(X)
write-lock1(Y)	

- T1 and T2 are in a deadlock
- But Site 1 knows only that T2 is waiting for T1; Site 2 knows only that T1 is waiting for T2

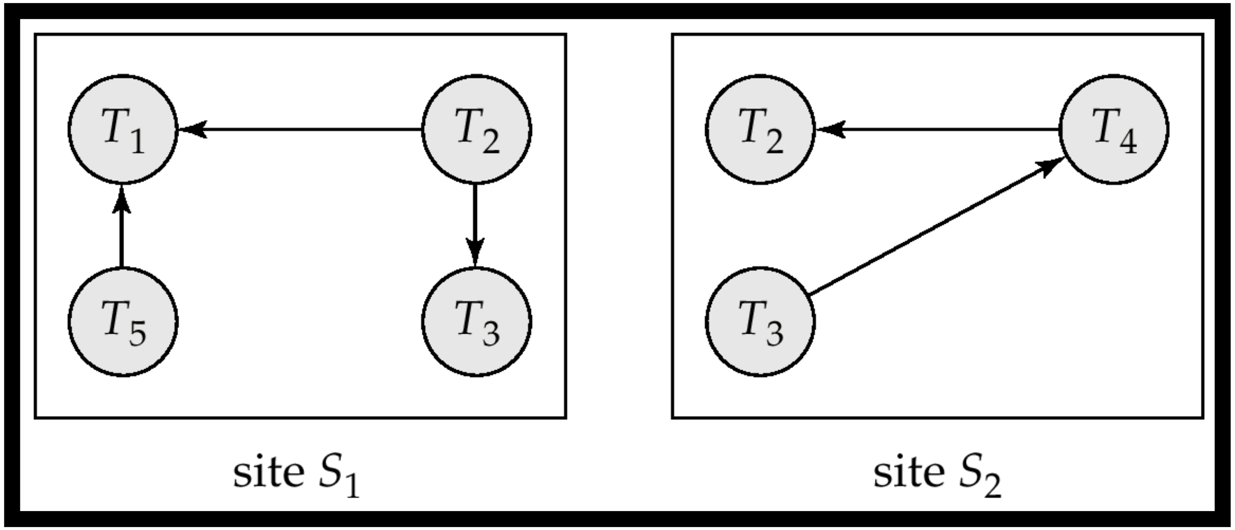
Distributed deadlocks

- Solution: a repetitive detection of deadlocks and abort of a transaction in a deadlock
- Management of a **wait-for-graph (WFG)** in each site
- A distributed protocol is run now and then to construct a **global WFG** which combines the local WFGs
- A deadlock exists when there is a **cycle in the global WFG**
- The global WFG is constructed in one site, called a **coordinator**

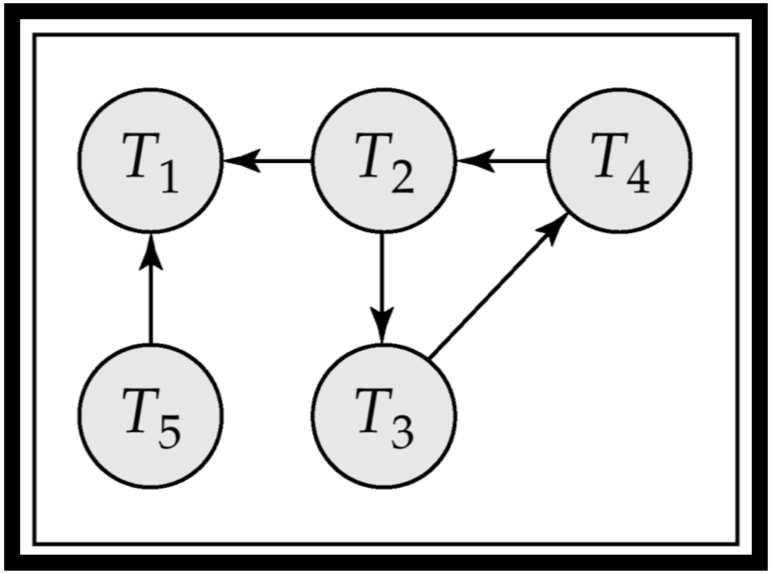
Distributed deadlocks

Example

Local WFGs



Global WFG



Distributed deadlocks

Protocol for the construction of the WFG

- Coordinator broadcasts a request for the local WFGs
- Coordinator combines the local WFGs to a global WFG and aborts a transaction in order to remove cycles in the global WFG
- Aborts are handled using the atomic commit protocol

Observation:

- The coordinator does not know the *actual* global WFG, only its *approximation* (because of communication delays)
- The consequence is the detection of cycles which do not exist at that time
- Hence, the coordinator sometimes unnecessarily aborts a transaction which is not involved in a deadlock

Distributed deadlocks

Example: phantom cycles in the WFG

- T2 releases the lock in S1: the edge T1→T2 should disappear from the WFG
- T2 asks for a lock which is has been assigned to T3 in S2: in site 2, the edge T2→T3 is added to the WFG
- However, the coordinator mat perceive the above actions in the different order, when it receives the WFG from S2 before receiving WFG from S1. Then it detects the cycle T1 → T2→T3 →T1 in its global WFG and unnecessarily aborts some of the transactions T1, T2, T3