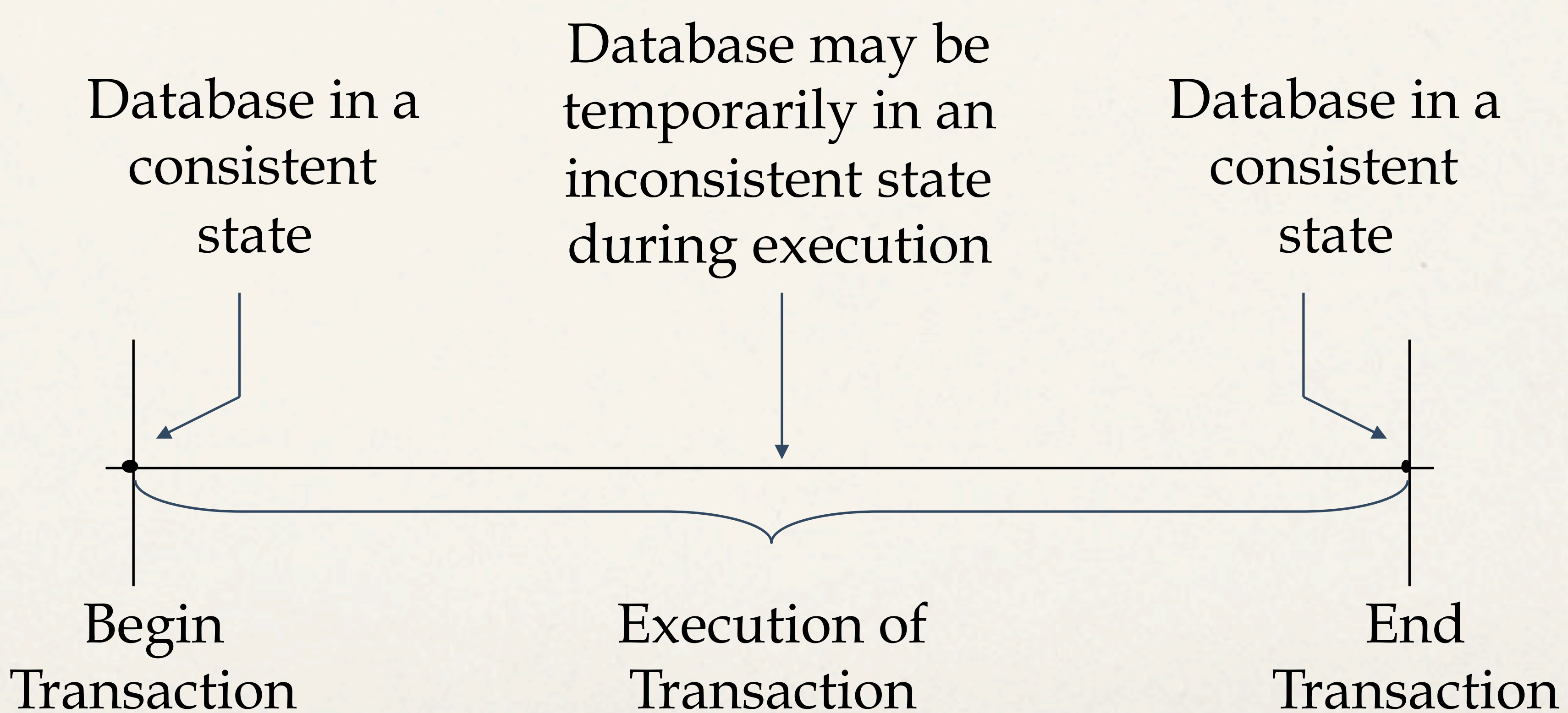
Outline

- Introduction
- Background
- Distributed Database Design
- Database Integration
- Semantic Data Control
- Distributed Query Processing
- Multidatabase Query Processing
- Distributed Transaction Management
  - ➡ Transaction Concepts and Models
  - ➡ Distributed Concurrency Control
  - ➡ Distributed Reliability
- Data Replication
- Parallel Database Systems
- Distributed Object DBMS
- Peer-to-Peer Data Management
- Web Data Management
- Current Issues

Transaction

A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.

- concurrency transparency
- failure transparency



Transaction Example – A Simple SQL Query

Transaction BUDGET_UPDATE

begin

```
EXEC SQL    UPDATE   PROJ
            SET      BUDGET = BUDGET*1.1
            WHERE    PNAME = “CAD/CAM”
```

end.

Example Database

Consider an airline reservation example with the relations:

FLIGHT(<u>FNO, DATE</u>, SRC, DEST, STSOLD, CAP)

CUST(<u>CNAME</u>, ADDR, BAL)

FC(<u>FNO, DATE, CNAME</u>,SPECIAL)

Example Transaction – SQL Version

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL  UPDATE    FLIGHT
                SET     STSOLD = STSOLD + 1
                WHERE   FNO = flight_no AND DATE = date;
    EXEC SQL  INSERT
                INTO    FC(FNO, DATE, CNAME, SPECIAL);
                VALUES  (flight_no, date, customer_name, null);
    output("reservation completed")
end . {Reservation}
```

Termination of Transactions

```
Begin_transaction Reservation
begin
    input(flight_no, date, customer_name);
    EXEC SQL    SELECT      STSOLD,CAP
                INTO        temp1,temp2
                FROM        FLIGHT
                WHERE       FNO = flight_no AND DATE =  date;
    if temp1 = temp2 then
        output("no free seats");
        Abort
    else
        EXEC SQL  UPDATE    FLIGHT
                    SET     STSOLD = STSOLD + 1
                    WHERE   FNO = flight_no AND DATE = date;
        EXEC SQL  INSERT
                    INTO    FC(FNO, DATE, CNAME, SPECIAL);
                    VALUES  (flight_no, date, customer_name, null);
       Commit
       output("reservation completed")
   endif
end . {Reservation}
```

Example Transaction – Reads & Writes

```
Begin_transaction Reservation
begin
        input(flight_no, date, customer_name);
        temp ← Read(flight_no(date).stsold);
        if temp = flight(date).cap then
        begin
            output("no free seats");
            Abort
        end
        else begin
            Write(flight(date).stsold, temp + 1);
            Write(flight(date).cname, customer_name);
            Write(flight(date).special, null);
            Commit;
            output("reservation completed")
        end
end. {Reservation}
```

Characterization

- Read set (RS)
  - ➡ The set of data items that are read by a transaction
- Write set (WS)
  - ➡ The set of data items whose values are changed by this transaction
- Base set (BS)
  - ➡ $RS \cup WS$

Formalization

Let

- ➡ $O_{ij}(x)$ be some operation O_j of transaction T_i operating on entity x, where $O_j \in$ {read,write} and O_j is atomic
- ➡ $OS_i = \bigcup_j O_{ij}$
- ➡ $N_i \in$ {abort,commit}

Transaction T_i is a partial order $T_i = \{\Sigma_i, <_i\}$ where

❶ $\Sigma_i = OS_i \cup \{N_i\}$

❷ For any two operations O_{ij} , $O_{ik} \in OS_i$, if $O_{ij} = R(x)$ and $O_{ik} = W(x)$ for any data item x, then either $O_{ij} <_i O_{ik}$ or $O_{ik} <_i O_{ij}$

❸ $\forall O_{ij} \in OS_i, O_{ij} <_i N_i$

Example

Consider a transaction T:

Read(x)

Read(y)

$x \leftarrow x + y$

Write(x)

Commit

Then

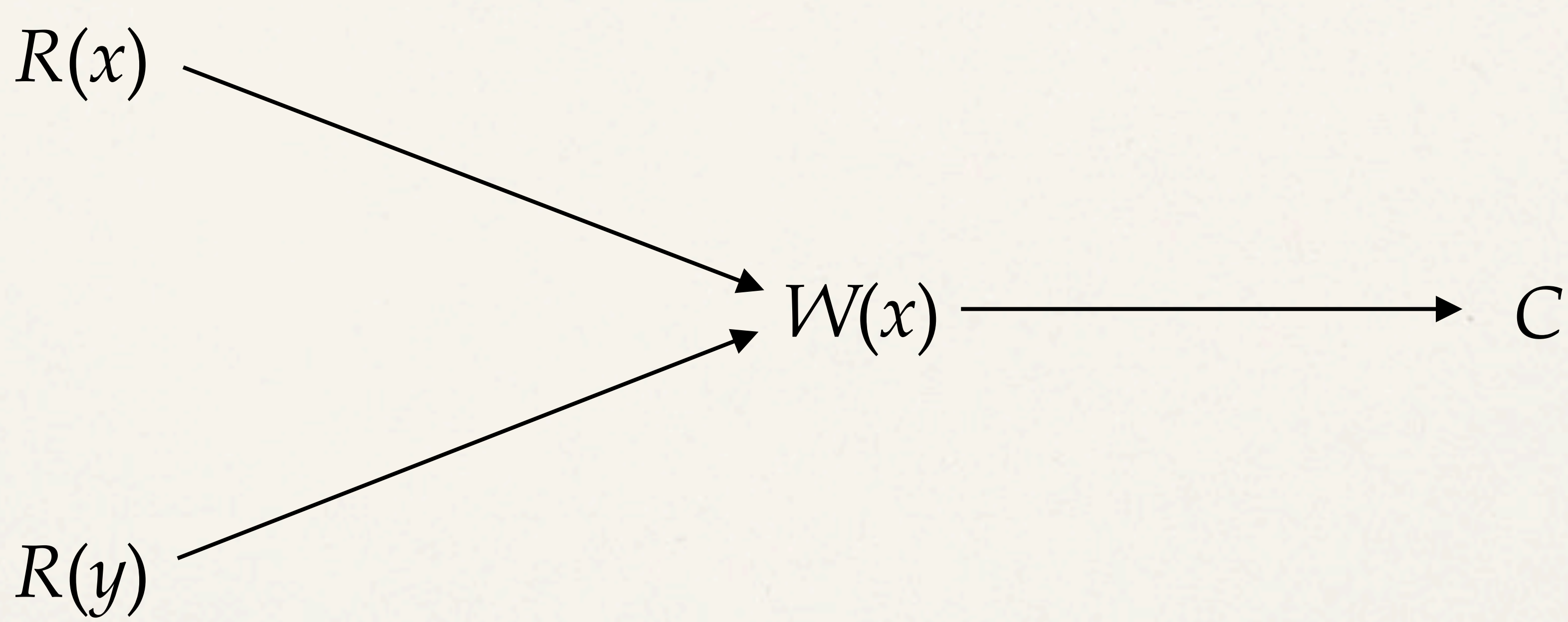
$\Sigma = \{R(x), R(y), W(x), C\}$

$< = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$

DAG Representation

Assume

$< \; = \{(R(x),W(x)),\ (R(y),W(x)),\ (W(x),\ C),\ (R(x),\ C),\ (R(y),\ C)\}$



Principles of Transactions

ATOMICITY

- all or nothing

CONSISTENCY

- no violation of integrity constraints

ISOLATION

- concurrent changes invisible $\Rightarrow$ serializable

DURABILITY

- committed updates persist

Atomicity

- Either all or none of the transaction's operations are performed.
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be undone.
- The activity of preserving the transaction's atomicity in presence of transaction aborts due to input errors, system overloads, or deadlocks is called transaction recovery.
- The activity of ensuring atomicity in the presence of system crashes is called crash recovery.

Consistency

- Internal consistency
  - ➡ A transaction which executes alone against a consistent database leaves it in a consistent state.
  - ➡ Transactions do not violate database integrity constraints.
- Transactions are correct programs

Consistency Degrees

- Degree 0
  - ➡ Transaction *T* does not overwrite dirty data of other transactions
  - ➡ Dirty data refers to data values that have been updated by a transaction prior to its commitment
- Degree 1
  - ➡ *T* does not overwrite dirty data of other transactions
  - ➡ *T* does not commit any writes before EOT

Consistency Degrees (cont'd)

- Degree 2
  - ➡ *T* does not overwrite dirty data of other transactions
  - ➡ *T* does not commit any writes before EOT
  - ➡ *T* does not read dirty data from other transactions
- Degree 3
  - ➡ *T* does not overwrite dirty data of other transactions
  - ➡ *T* does not commit any writes before EOT
  - ➡ *T* does not read dirty data from other transactions
  - ➡ Other transactions do not dirty any data read by *T* before *T* completes.

Isolation

- Serializability
  - ➡ If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.
- Incomplete results
  - ➡ An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - ➡ Necessary to avoid cascading aborts.

Isolation Example

- Consider the following two transactions:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x+1$		$x \leftarrow x+1$
	Write(x)		Write(x)
	Commit		Commit

- Possible execution sequences:

T_1:	Read(x)	T_1:	Read(x)
T_1:	$x \leftarrow x+1$	T_1:	$x \leftarrow x+1$
T_1:	Write(x)	T_2:	Read(x)
T_1:	Commit	T_1:	Write(x)
T_2:	Read(x)	T_2:	$x \leftarrow x+1$
T_2:	$x \leftarrow x+1$	T_2:	Write(x)
T_2:	Write(x)	T_1:	Commit
T_2:	Commit	T_2:	Commit

SQL-92 Isolation Levels

Phenomena:

- Dirty read
  - T_1 modifies x which is then read by T_2 before T_1 terminates; T_1 aborts $\Rightarrow$ T_2 has read value which never exists in the database.
- Non-repeatable (fuzzy) read
  - T_1 reads x; T_2 then modifies or deletes x and commits. T_1 tries to read x again but reads a different value or can't find it.
- Phantom
  - T_1 searches the database according to a predicate while T_2 inserts new tuples that satisfy the predicate.

SQL-92 Isolation Levels (cont'd)

- Read Uncommitted
  - ➡ For transactions operating at this level, all three phenomena are possible.
- Read Committed
  - ➡ Fuzzy reads and phantoms are possible, but dirty reads are not.
- Repeatable Read
  - ➡ Only phantoms possible.
- Anomaly Serializable
  - ➡ None of the phenomena are possible.

Durability

- Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.
- Database recovery

Characterization of Transactions

- Based on
  - Application areas
    - Non-distributed vs. distributed
    - Compensating transactions
    - Heterogeneous transactions
  - Timing
    - On-line (short-life) vs batch (long-life)
  - Organization of read and write actions
    - Two-step
    - Restricted
    - Action model
  - Structure
    - Flat (or simple) transactions
    - Nested transactions
    - Workflows

Transaction Structure

- Flat transaction
  - ➡ Consists of a sequence of primitive operations embraced between a **begin** and **end** markers.

    Begin_transaction Reservation

    …

    end.
- Nested transaction
  - ➡ The operations of a transaction may themselves be transactions.

    Begin_transaction Reservation

    …

    Begin_transaction Airline

    …

    end. {Airline}

    Begin_transaction Hotel

    …

    end. {Hotel}

    end. {Reservation}

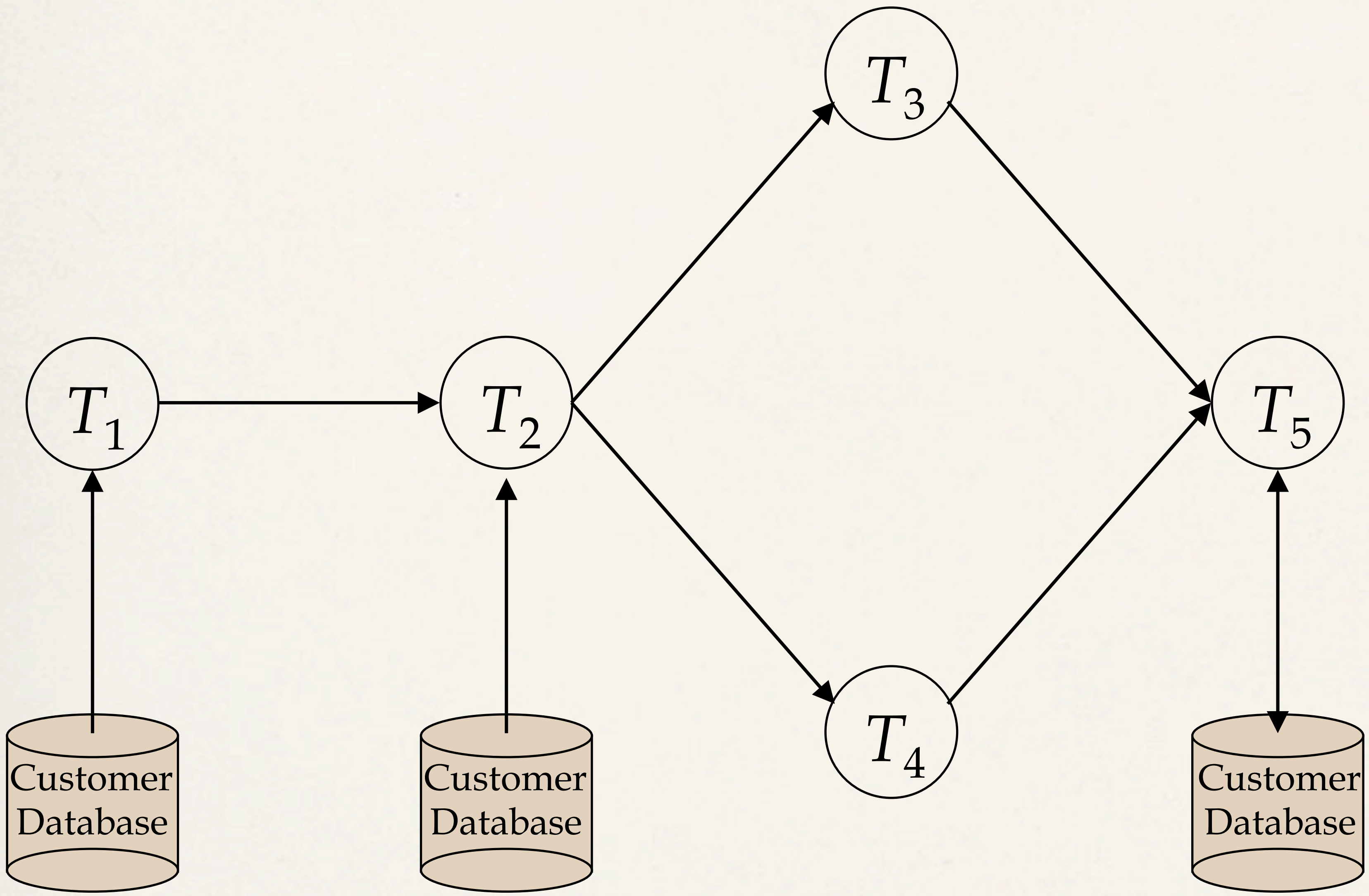
Nested Transactions

- Have the same properties as their parents ➪ may themselves have other nested transactions.
- Introduces concurrency control and recovery concepts to within the transaction.
- Types
  - ➡ Closed nesting
    - ✦ Subtransactions begin after their parents and finish before them.
    - ✦ Commitment of a subtransaction is conditional upon the commitment of the parent (commitment through the root).
  - ➡ Open nesting
    - ✦ Subtransactions can execute and commit independently.
    - ✦ Compensation may be necessary.

Workflows

- "A collection of tasks organized to accomplish some business process."
- Types
  - ➡ Human-oriented workflows
    - ✦ Involve humans in performing the tasks.
    - ✦ System support for collaboration and coordination; but no system-wide consistency definition
  - ➡ System-oriented workflows
    - ✦ Computation-intensive & specialized tasks that can be executed by a computer
    - ✦ System support for concurrency control and recovery, automatic task execution, notification, etc.
  - ➡ Transactional workflows
    - ✦ In between the previous two; may involve humans, require access to heterogeneous, autonomous and/or distributed systems, and support selective use of ACID properties

Workflow Example



T_1: Customer request obtained

T_2: Airline reservation performed

T_3: Hotel reservation performed

T_4: Auto reservation performed

T_5: Bill generated

Transactions Provide…

- *Atomic* and *reliable* execution in the presence of failures
- *Correct* execution in the presence of multiple user accesses
- Correct management of *replicas* (if they support it)

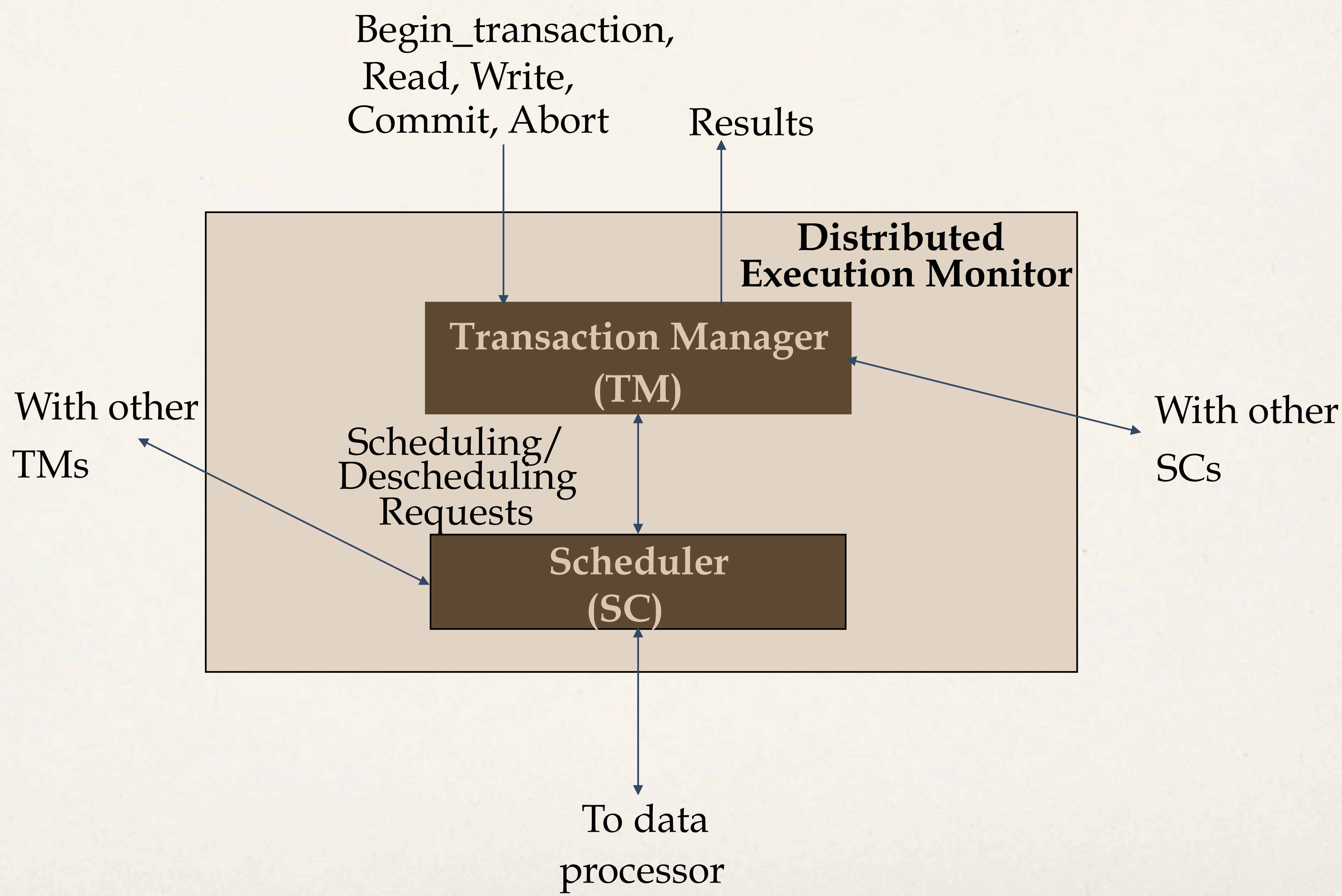
Transaction Processing Issues

- Transaction structure (usually called transaction model)
  - ➡ Flat (simple), nested
- Internal database consistency
  - ➡ Semantic data control (integrity enforcement) algorithms
- Reliability protocols
  - ➡ Atomicity & Durability
  - ➡ Local recovery protocols
  - ➡ Global commit protocols

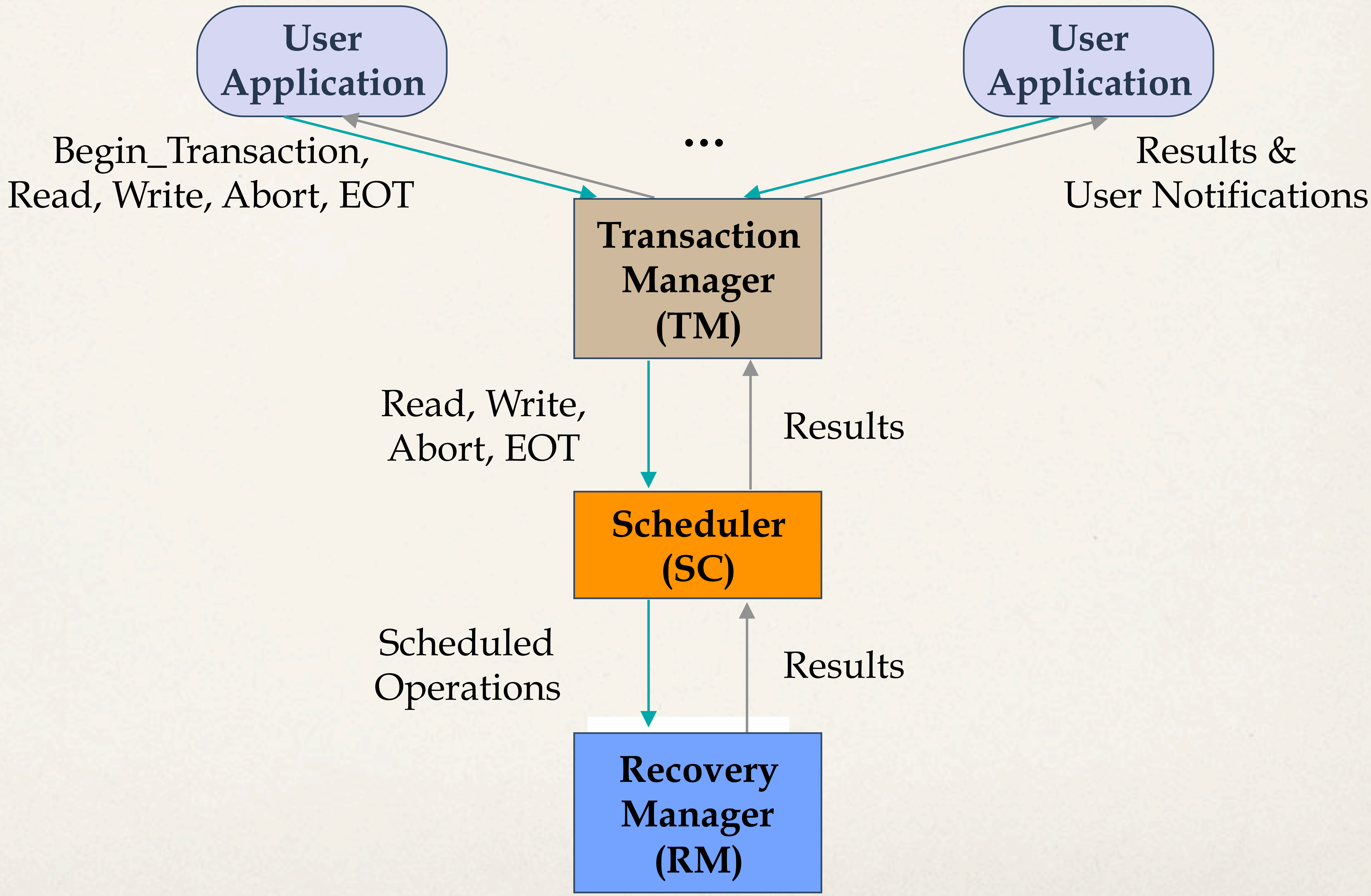
Transaction Processing Issues

- Concurrency control algorithms
  - ➡ How to synchronize concurrent transaction executions (correctness criterion)
  - ➡ Intra-transaction consistency, Isolation
- Replica control protocols
  - ➡ How to control the mutual consistency of replicated data
  - ➡ One copy equivalence and ROWA

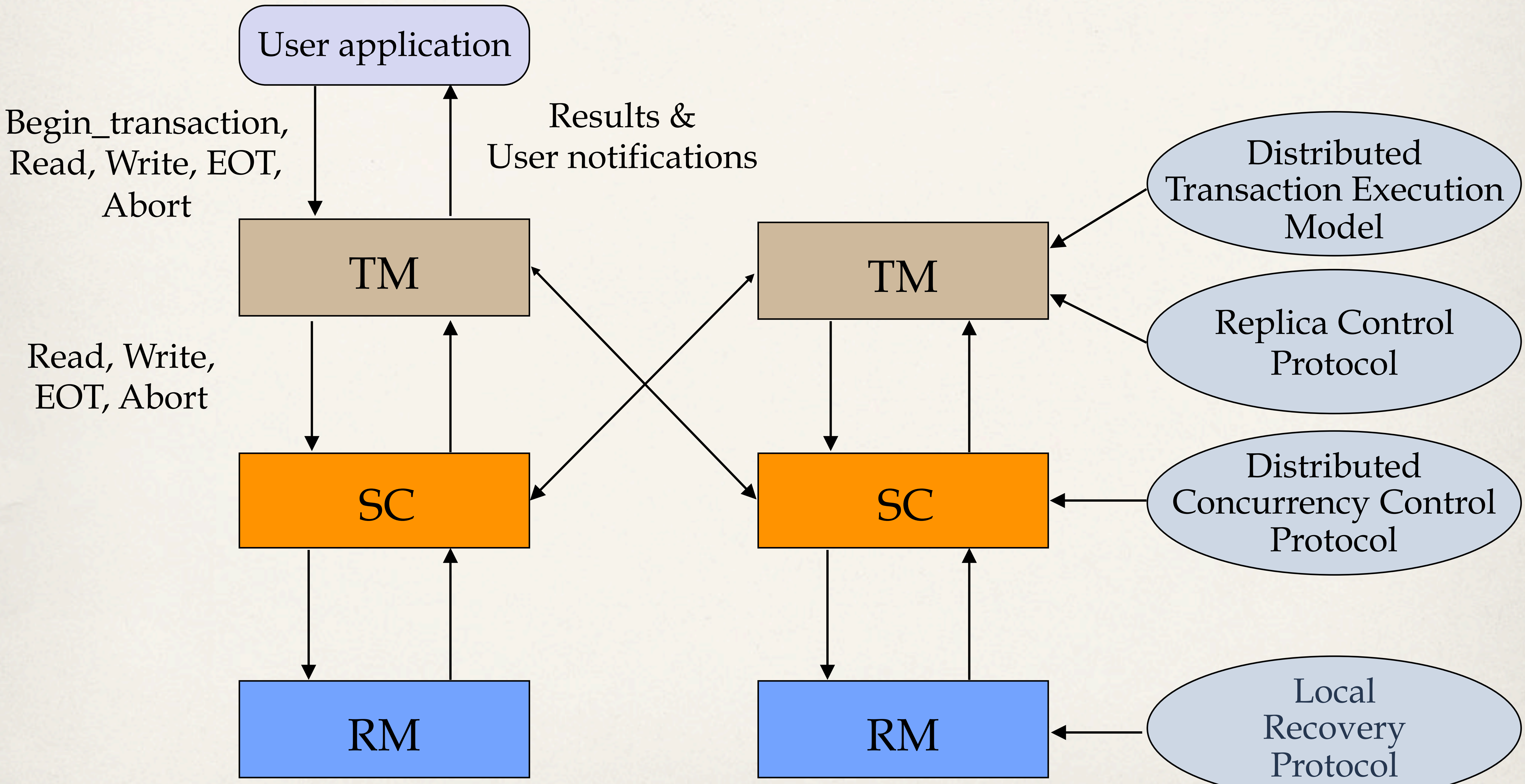
Architecture Revisited



Centralized Transaction Execution



Distributed Transaction Execution



Outline

- Introduction
- Background
- Distributed Database Design
- Database Integration
- Semantic Data Control
- Distributed Query Processing
- Multidatabase Query Processing
- Distributed Transaction Management
  - ➡ Transaction Concepts and Models
  - ➡ Distributed Concurrency Control
  - ➡ Distributed Reliability
- Data Replication
- Parallel Database Systems
- Distributed Object DBMS
- Peer-to-Peer Data Management
- Web Data Management
- Current Issues

Concurrency Control

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.
- Anomalies:
  - ➡ Lost updates
    - ✦ The effects of some transactions are not reflected on the database.
  - ➡ Inconsistent retrievals
    - ✦ A transaction, if it reads the same data item more than once, should always read the same value.

Execution History (or Schedule)

- An order in which the operations of a set of transactions are executed.
- A history (schedule) can be defined as a partial order over the operations of a set of transactions.

T_1: Read(x)	T_2: Write(x)	T_3: Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$$

Formalization of History

A complete history over a set of transactions $T=\{T_1, \ldots, T_n\}$ is a partial order $H_c(T) = \{\Sigma_T, <_H\}$ where

❶ $\Sigma_T = \bigcup_i \Sigma_i$, for $i = 1, 2, \ldots, n$

❷ $<_H \subseteq \bigcup_i <_{T_i}$, for $i = 1, 2, \ldots, n$

❸ For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} <_H O_{kl}$ or $O_{kl} <_H O_{ij}$

Complete Schedule – Example

Given three transactions

T_1:	Read(x)	T_2:	Write(x)	T_3:	Read(x)
	Write(x)		Write(y)		Read(y)
	Commit		Read(z)		Read(z)
			Commit		Commit

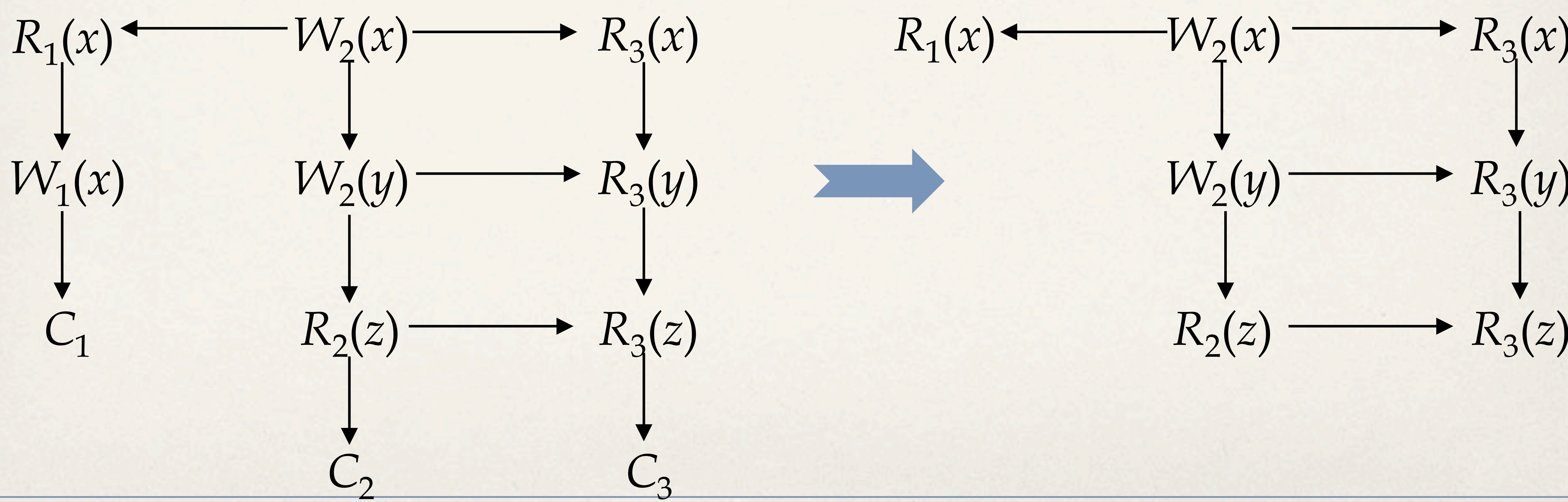
A possible complete schedule is given as the DAG

$$\begin{tikzcd}
R_1(x) \arrow[d] & W_2(x) \arrow[l] \arrow[r] \arrow[d] & R_3(x) \arrow[d] \\
W_1(x) \arrow[d] & W_2(y) \arrow[r] \arrow[d] & R_3(y) \arrow[d] \\
C_1 & R_2(z) \arrow[r] \arrow[d] & R_3(z) \arrow[d] \\
 & C_2 & C_3
\end{tikzcd}$$

Schedule Definition

A schedule is a prefix of a complete schedule such that only some of the operations and only some of the ordering relationships are included.

T_1:	Read(x)	T_2:	Write(x)	T_3:	Read(x)
	Write(x)		Write(y)		Read(y)
	Commit		Read(z)		Read(z)
			Commit		Commit



Serial History

- All the actions of a transaction occur consecutively.
- No interleaving of transaction operations.
- If each transaction is consistent (obeys integrity rules), then the database is guaranteed to be consistent at the end of executing a serial history.

T_1: Read(x)	T_2: Write(x)	T_3: Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H = \{\underbrace{W_2(x), W_2(y), R_2(z)}_{T_2}, \underbrace{R_1(x), W_1(x)}_{T_1}, \underbrace{R_3(x), R_3(y), R_3(z)}_{T_3}\}$$

Serializable History

- Transactions execute concurrently, but the net effect of the resulting history upon the database is equivalent to some serial history.
- Equivalent with respect to what?
  - ➡ ***Conflict equivalence***: the relative order of execution of the conflicting operations belonging to unaborted transactions in two histories are the same.
  - ➡ ***Conflicting operations***: two incompatible operations (e.g., Read and Write) conflict if they both access the same data item.
    - ✦ Incompatible operations of each transaction is assumed to conflict; do not change their execution orders.
    - ✦ If two operations from two different transactions conflict, the corresponding transactions are also said to conflict.

Serializable History

T_1: Read(x)
Write(x)
Commit

T_2: Write(x)
Write(y)
Read(z)
Commit

T_3: Read(x)
Read(y)
Read(z)
Commit

The following are not conflict equivalent

$H_s=\{W_2(x),W_2(y),R_2(z),R_1(x),W_1(x),R_3(x),R_3(y),R_3(z)\}$

$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),W_2(y),R_3(y),R_2(z),R_3(z)\}$

The following are conflict equivalent; therefore H_2 is *serializable*.

$H_s=\{W_2(x),W_2(y),R_2(z),R_1(x),W_1(x),R_3(x),R_3(y),R_3(z)\}$

$H_2=\{W_2(x),R_1(x),W_1(x),R_3(x),W_2(y),R_3(y),R_2(z),R_3(z)\}$

Serializability in Distributed DBMS

- Somewhat more involved. Two histories have to be considered:
  - ➡ local histories
  - ➡ global history
- For global transactions (i.e., global history) to be serializable, two conditions are necessary:
  - ➡ Each local history should be serializable.
  - ➡ Two conflicting operations should be in the same relative order in all of the local histories where they appear together.

Global Non-serializability

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x$-100		Read(y)
	Write(x)		Commit
	Read(y)		
	$y \leftarrow y$+100		
	Write(y)		
	Commit		

- x stored at Site 1, y stored at Site 2
- LH_1, LH_2 are individually serializable (in fact serial), but the two transactions are not globally serializable.

$$LH_1=\{R_1(x),W_1(x), R_2(x)\}$$

$$LH_2=\{R_2(y), R_1(y),W_1(y)\}$$

Concurrency Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
    - Centralized (primary site) 2PL
    - Primary copy 2PL
    - Distributed 2PL
  - Timestamp Ordering (TO)
    - Basic TO
    - Multiversion TO
    - Conservative TO
  - Hybrid
- Optimistic
  - Locking-based
  - Timestamp ordering-based

Locking-Based Algorithms

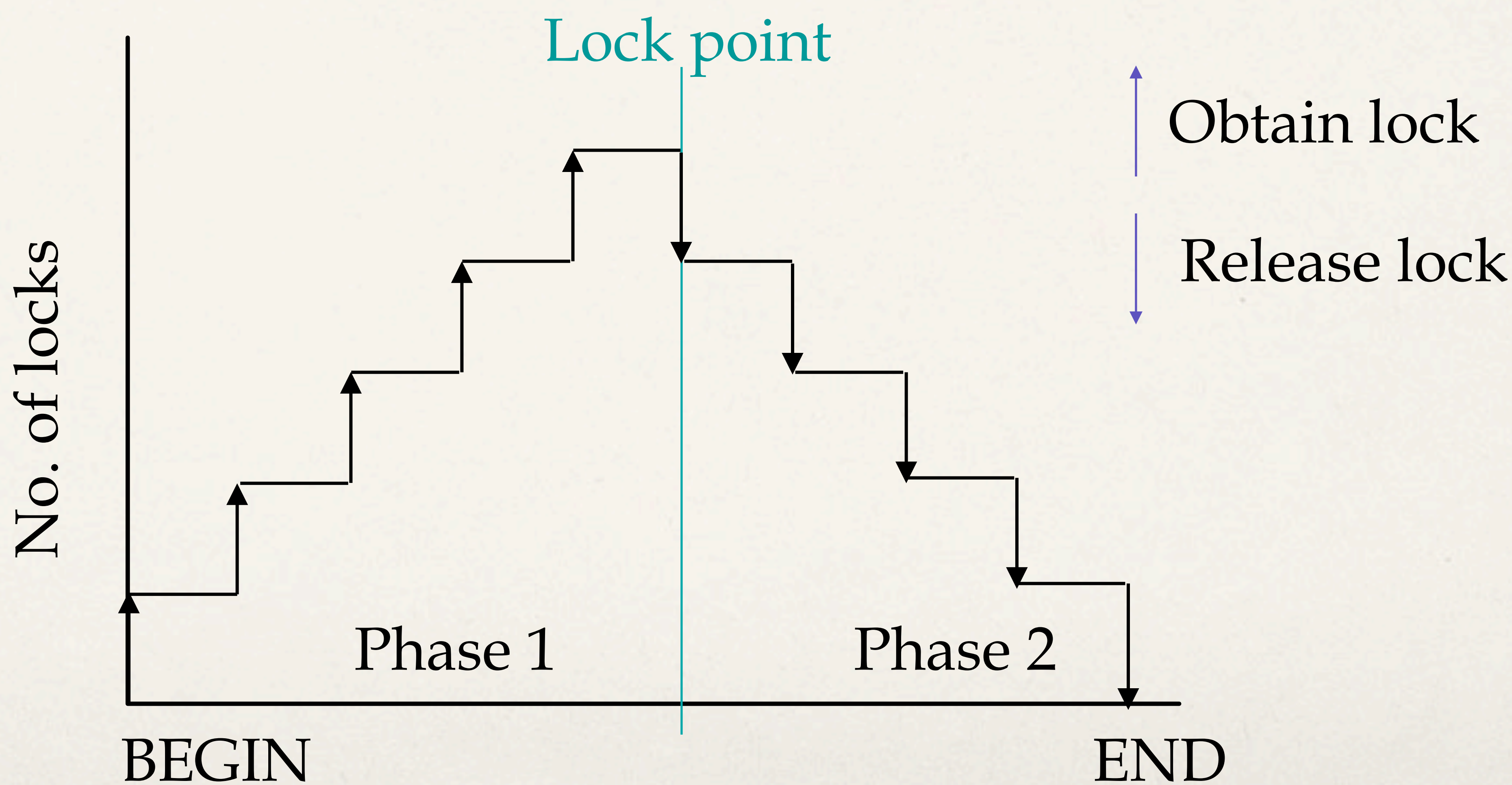
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

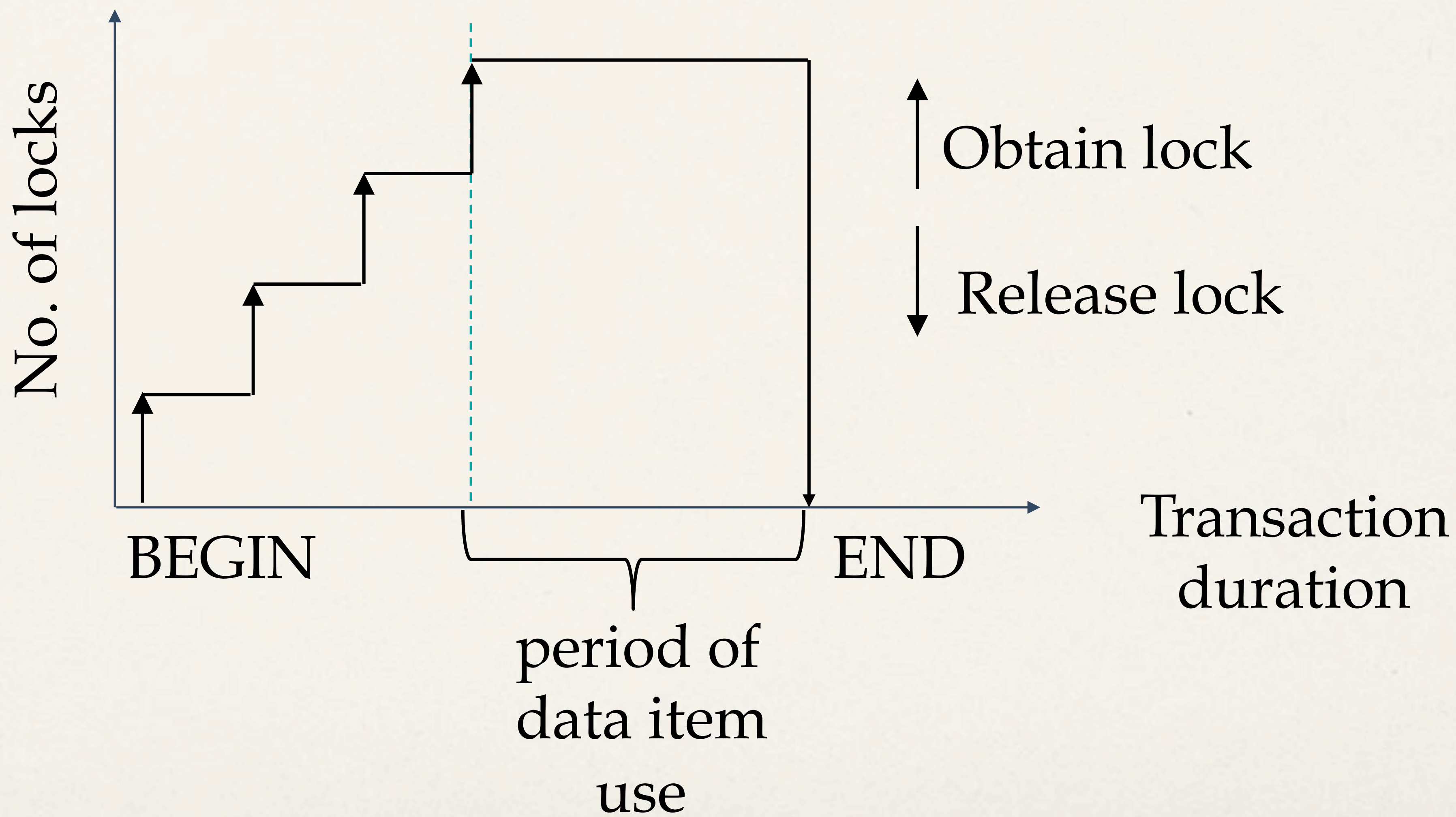
Two-Phase Locking (2PL)

1. A Transaction locks an object before using it.
2. When an object is locked by another transaction, the requesting transaction must wait.
3. When a transaction releases a lock, it may not request another lock.



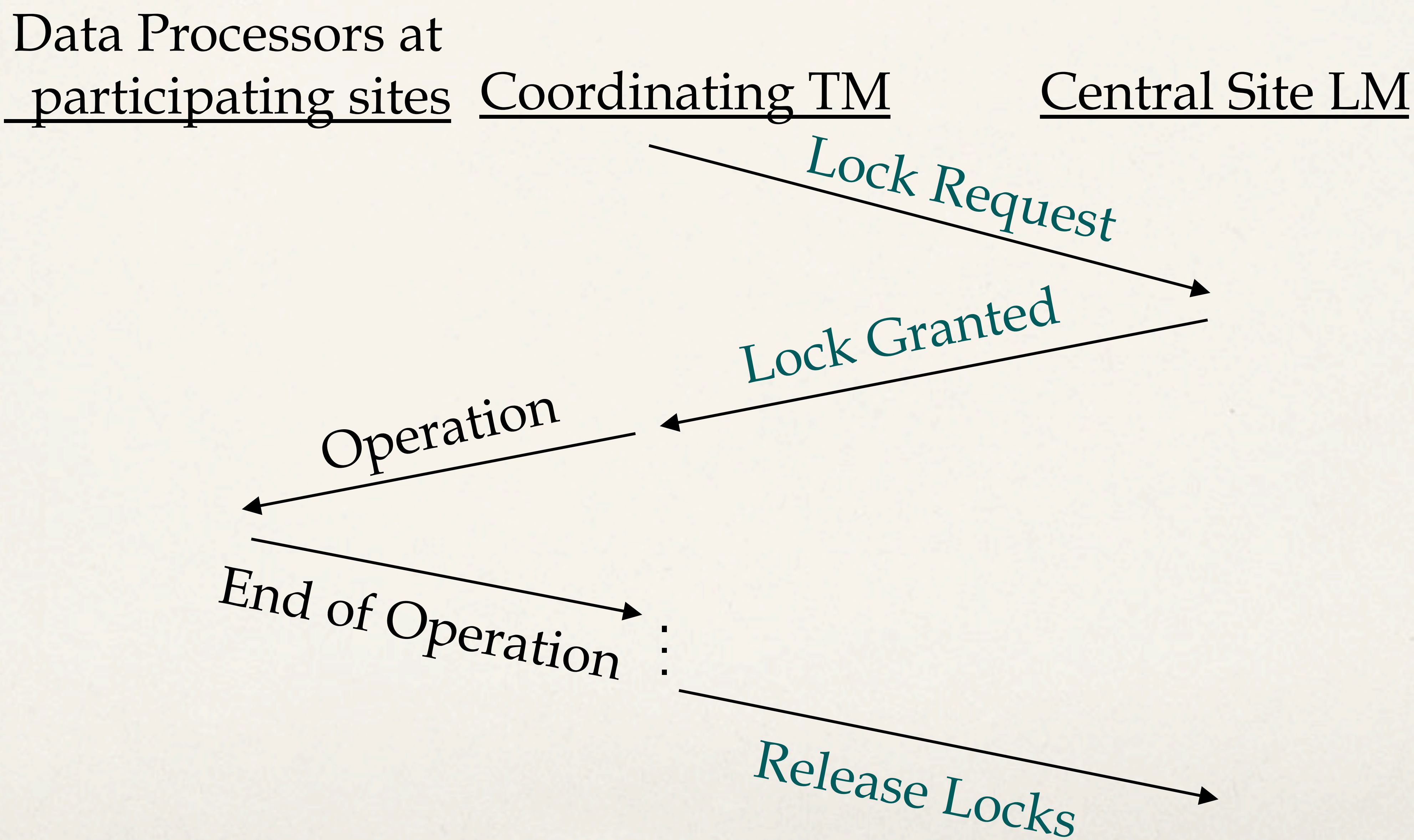
Strict 2PL

Hold locks until the end.



Centralized 2PL

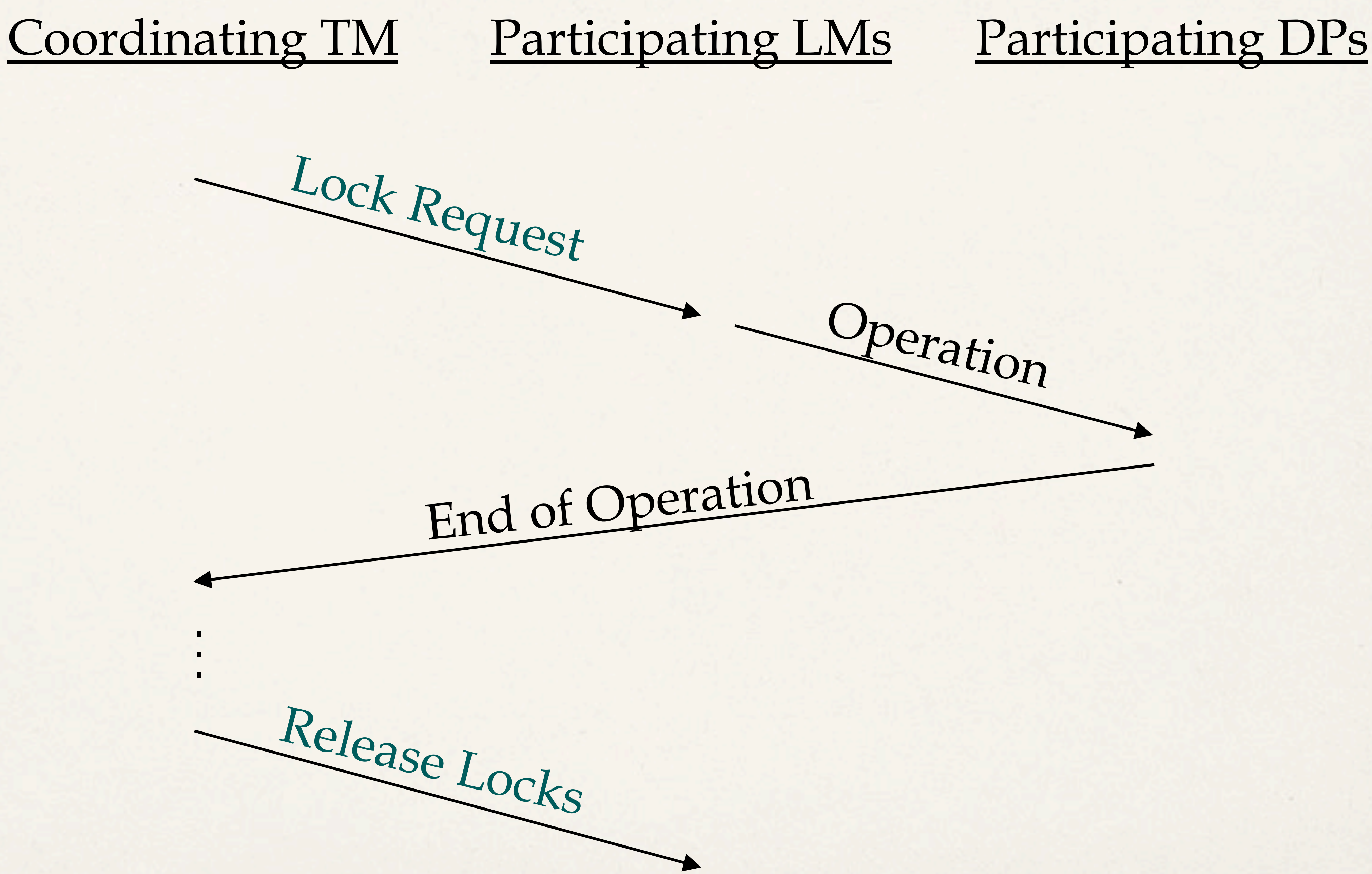
- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



Distributed 2PL

- 2PL schedulers are placed at each site. Each scheduler handles lock requests for data at that site.
- A transaction may read any of the replicated copies of item x, by obtaining a read lock on one of the copies of x. Writing into x requires obtaining write locks for all copies of x.

Distributed 2PL Execution



Timestamp Ordering

❶ Transaction (T_i) is assigned a globally unique timestamp $ts(T_i)$.

❷ Transaction manager attaches the timestamp to all operations issued by the transaction.

❸ Each data item is assigned a write timestamp (wts) and a read timestamp (rts):

- ➡ $rts(x)$ = largest timestamp of any read on x
- ➡ $wts(x)$ = largest timestamp of any read on x

❹ Conflicting operations are resolved by timestamp order.

Basic T/O:

<u>for $R_i(x)$</u>	<u>for $W_i(x)$</u>
if $ts(T_i) < wts(x)$	**if** $ts(T_i) < rts(x)$ **and** $ts(T_i) < wts(x)$
then reject $R_i(x)$	**then** reject $W_i(x)$
else accept $R_i(x)$	**else** accept $W_i(x)$
$rts(x) \leftarrow ts(T_i)$	$wts(x) \leftarrow ts(T_i)$

Conservative Timestamp Ordering

- Basic timestamp ordering tries to execute an operation as soon as it receives it
  - ➡ progressive
  - ➡ too many restarts since there is no delaying
- Conservative timestamping delays each operation until there is an assurance that it will not be restarted
- Assurance?
  - ➡ No other operation with a smaller timestamp can arrive at the scheduler
  - ➡ Note that the delay may result in the formation of deadlocks

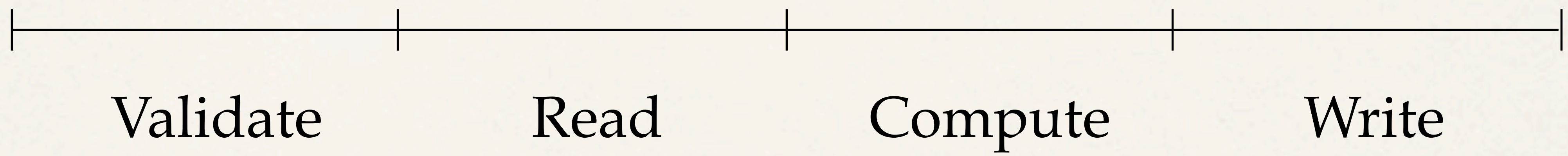
Multiversion Timestamp Ordering

- Do not modify the values in the database, create new values.
- A $R_i(x)$ is translated into a read on one version of x.
  - ➡ Find a version of x (say x_v) such that $ts(x_v)$ is the largest timestamp less than $ts(T_i)$.
- A $W_i(x)$ is translated into $W_i(x_w)$ and accepted if the scheduler has not yet processed any $R_j(x_r)$ such that

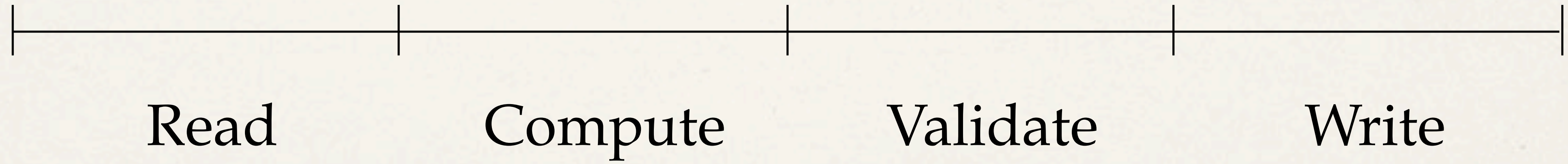
$$ts(T_i) < ts(x_r) < ts(T_j)$$

Optimistic Concurrency Control Algorithms

Pessimistic execution



Optimistic execution

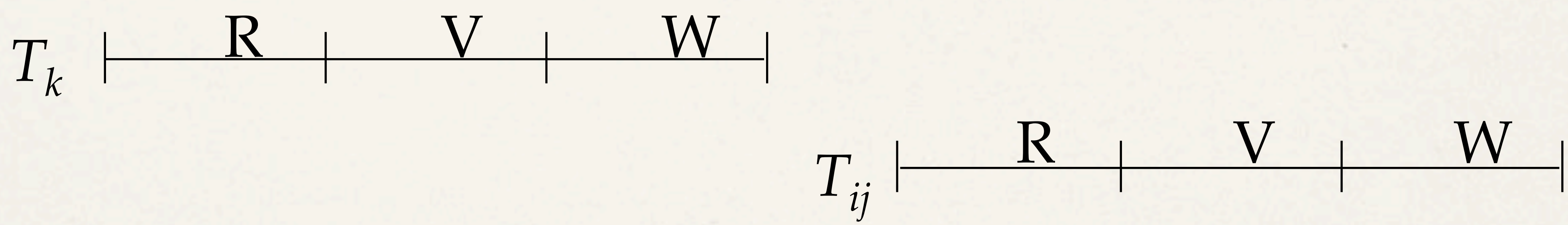


Optimistic Concurrency Control Algorithms

- Transaction execution model: divide into subtransactions each of which execute at a site
  - T_{ij}: transaction T_i that executes at site j
- Transactions run independently at each site until they reach the end of their read phases
- All subtransactions are assigned a timestamp at the end of their read phase
- Validation test performed during validation phase. If one fails, all rejected.

Optimistic CC Validation Test

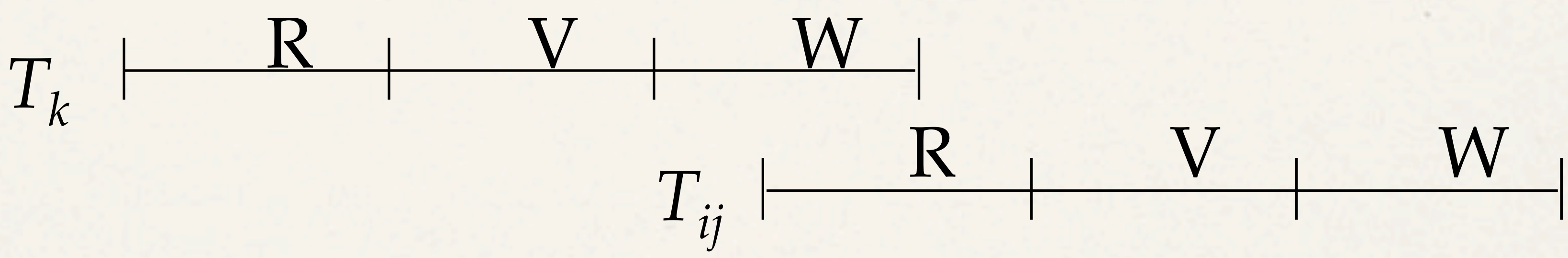
1. If all transactions T_k where $ts(T_k) < ts(T_{ij})$ have completed their write phase before T_{ij} has started its read phase, then validation succeeds
   - Transaction executions in serial order



Optimistic CC Validation Test

❷ If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$ and which completes its write phase while T_{ij} is in its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \varnothing$

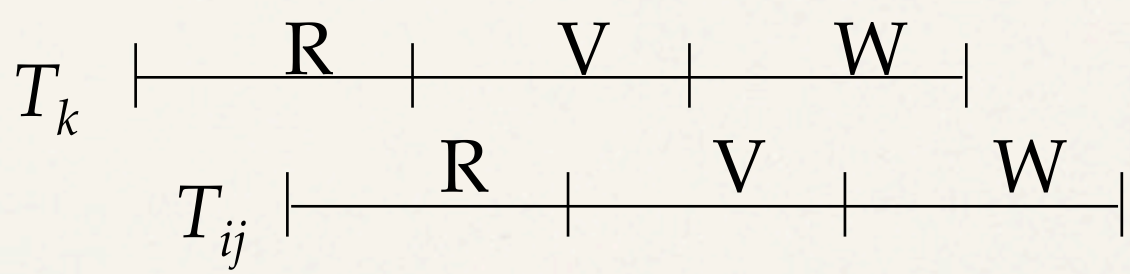
- ➡ Read and write phases overlap, but T_{ij} does not read data items written by T_k



Optimistic CC Validation Test

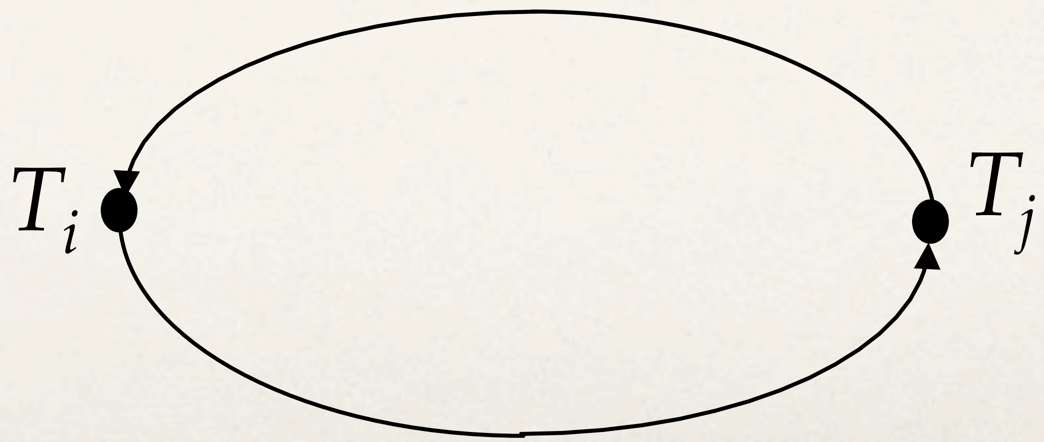
❸ If there is any transaction T_k such that $ts(T_k) < ts(T_{ij})$ and which completes its read phase before T_{ij} completes its read phase, then validation succeeds if $WS(T_k) \cap RS(T_{ij}) = \varnothing$ and $WS(T_k) \cap WS(T_{ij}) = \varnothing$

- ➡ They overlap, but don't access any common data items.



Deadlock

- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- TO-based algorithms that involve waiting may cause deadlocks.
- Wait-for graph
  - ➡ If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.

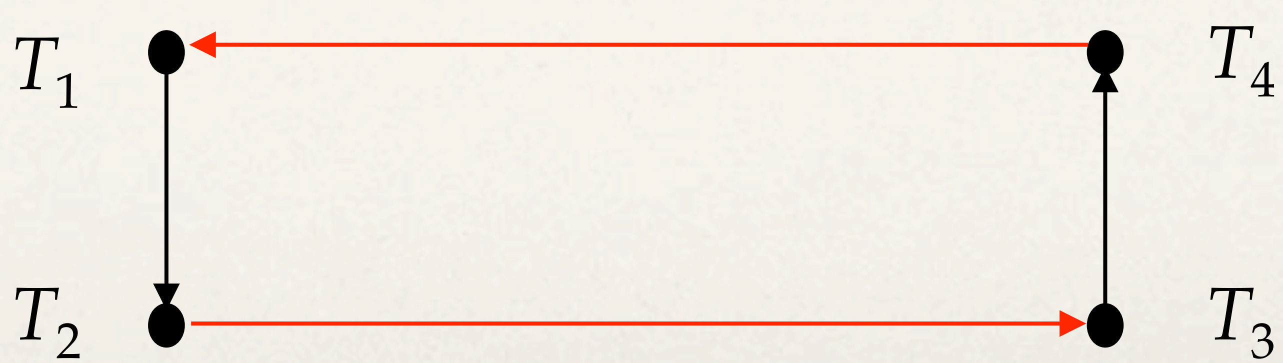


Local versus Global WFG

Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2. Also assume T_3 waits for a lock held by T_4 which waits for a lock held by T_1 which waits for a lock held by T_2 which, in turn, waits for a lock held by T_3.

Local WFG

Global WFG



Deadlock Management

- Ignore
  - ➡ Let the application programmer deal with it, or restart the system
- Prevention
  - ➡ Guaranteeing that deadlocks can never occur in the first place. Check transaction when it is initiated. Requires no run time support.
- Avoidance
  - ➡ Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support.
- Detection and Recovery
  - ➡ Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.

Deadlock Prevention

- All resources which may be needed by a transaction must be predeclared.
  - ➡ The system must guarantee that none of the resources will be needed by an ongoing transaction.
  - ➡ Resources must only be reserved, but not necessarily allocated a priori
  - ➡ Unsuitability of the scheme in database environment
  - ➡ Suitable for systems that have no provisions for undoing processes.
- Evaluation:
  - – Reduced concurrency due to preallocation
  - – Evaluating whether an allocation is safe leads to added overhead.
  - – Difficult to determine (partial order)
  - \+ No transaction rollback or restart is involved.

Deadlock Avoidance

- Transactions are not required to request resources a priori.
- Transactions are allowed to proceed unless a requested resource is unavailable.
- In case of conflict, transactions may be allowed to wait for a fixed time interval.
- Order either the data items or the sites and always request locks in that order.
- More attractive than prevention in a database environment.

Deadlock Avoidance – Wait-Die Algorithm

If T_i requests a lock on a data item which is already locked by T_j, then T_i is permitted to wait iff $ts(T_i)<ts(T_j)$. If $ts(T_i)>ts(T_j)$, then T_i is aborted and restarted with the same timestamp.

- **if** $ts(T_i)<ts(T_j)$ **then** T_i waits **else** T_i dies
- non-preemptive: T_i never preempts T_j
- prefers younger transactions

Deadlock Avoidance – Wound-Wait Algorithm

If T_i requests a lock on a data item which is already locked by T_j , then T_i is permitted to wait iff $ts(T_i)>ts(T_j)$. If $ts(T_i)<ts(T_j)$, then T_j is aborted and the lock is granted to T_i.

- **if** $ts(T_i)<ts(T_j)$ **then** T_j is wounded **else** T_i waits
- preemptive: T_i preempts T_j if it is younger
- prefers older transactions

Deadlock Detection

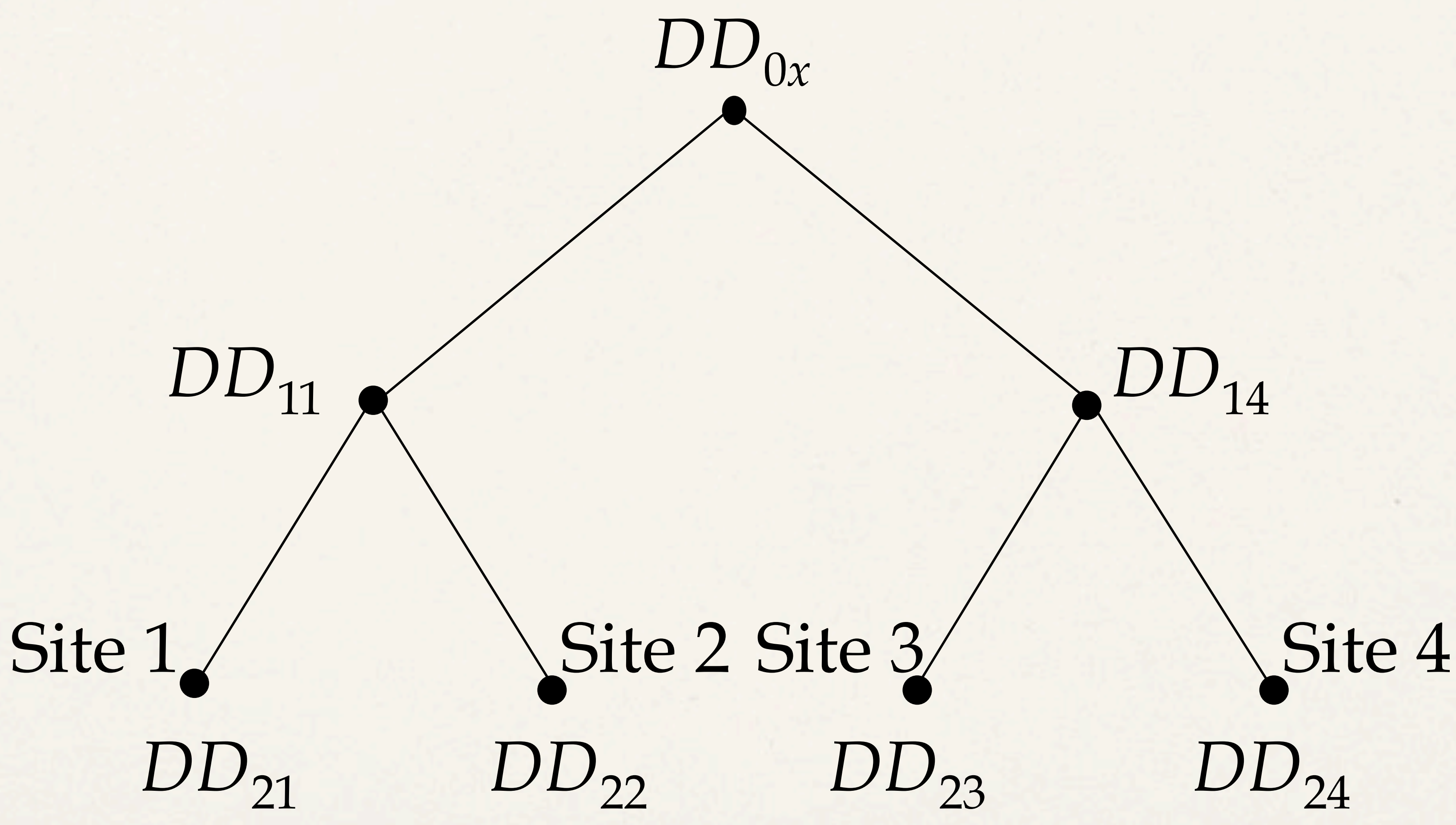
- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - ➡ Centralized
  - ➡ Distributed
  - ➡ Hierarchical

Centralized Deadlock Detection

- One site is designated as the deadlock detector for the system. Each scheduler periodically sends its local WFG to the central site which merges them to a global WFG to determine cycles.
- How often to transmit?
  - ➡ Too often ⇒ higher communication cost but lower delays due to undetected deadlocks
  - ➡ Too late ⇒ higher delays due to deadlocks, but lower communication cost
- Would be a reasonable choice if the concurrency control algorithm is also centralized.
- Proposed for Distributed INGRES

Hierarchical Deadlock Detection

Build a hierarchy of detectors



Distributed Deadlock Detection

- Sites cooperate in detection of deadlocks.
- One example:
  - ➡ The local WFGs are formed at each site and passed on to other sites. Each local WFG is modified as follows:
    1. Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    2. The edges in the local WFG which show that local transactions are waiting for transactions at other sites are joined with edges in the local WFGs which show that remote transactions are waiting for local ones.
  - ➡ Each local deadlock detector:
    - ✦ looks for a cycle that does not involve the external edge. If it exists, there is a local deadlock which can be handled locally.
    - ✦ looks for a cycle involving the external edge. If it exists, it indicates a potential global deadlock. Pass on the information to the next site.

“Relaxed” Concurrency Control

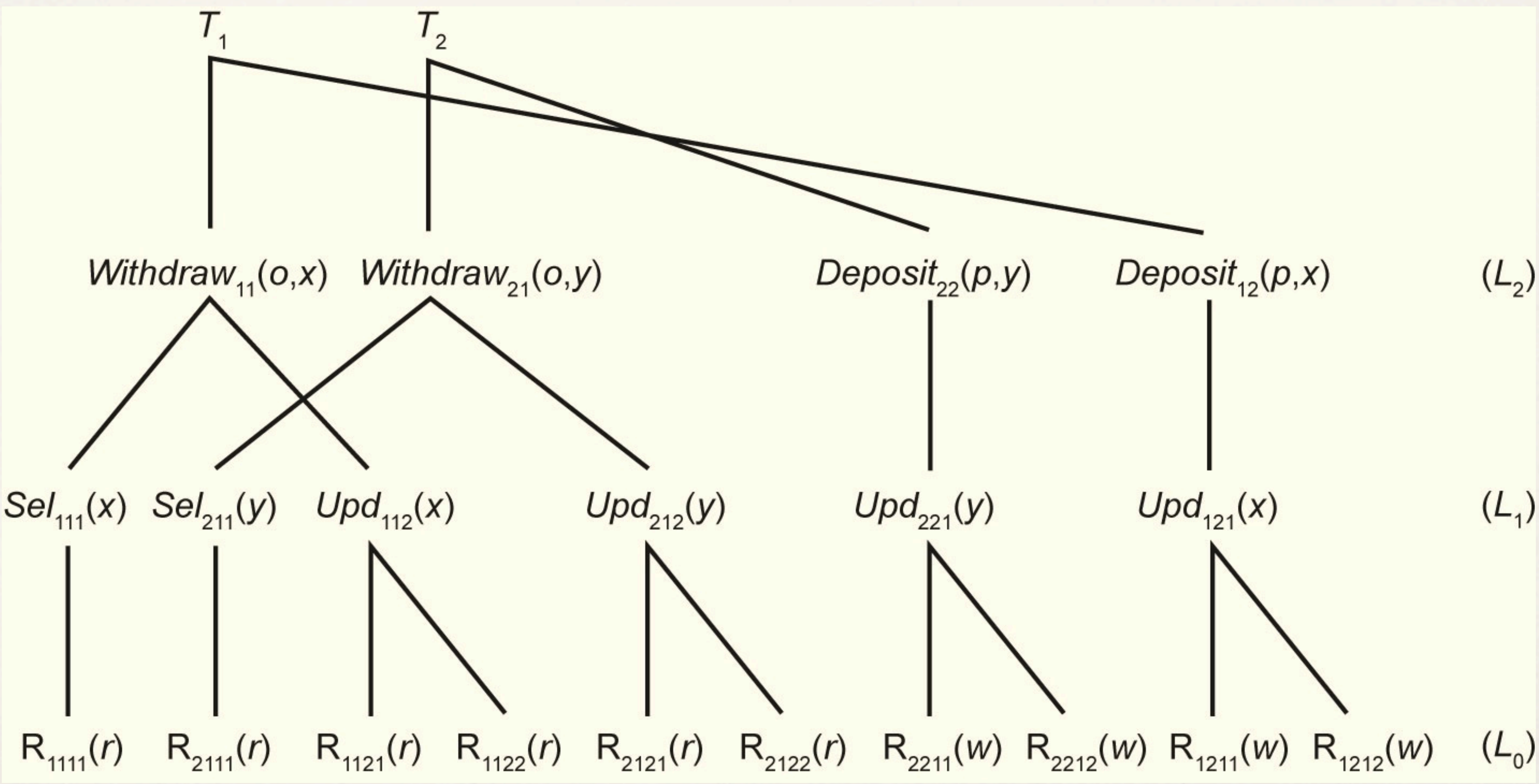
- Non-serializable histories
  - ➡ E.g., ordered shared locks
  - ➡ Semantics of transactions can be used
    - ✦ Look at semantic compatibility of operations rather than simply looking at reads and writes
- Nested distributed transactions
  - ➡ Closed nested transactions
  - ➡ Open nested transactions
  - ➡ Multilevel transactions

Multilevel Transactions

Consider two transactions

T_1: Withdraw(o,x)
Deposit(p,x)

T_2: Withdraw(o,y)
Deposit(p,y)



Outline

- Introduction
- Background
- Distributed Database Design
- Database Integration
- Semantic Data Control
- Distributed Query Processing
- Distributed Transaction Management
  - ➡ Transaction Concepts and Models
  - ➡ Distributed Concurrency Control
  - ➡ Distributed Reliability
- Data Replication
- Parallel Database Systems
- Distributed Object DBMS
- Peer-to-Peer Data Management
- Web Data Management
- Current Issues

Reliability

Problem:

How to maintain

atomicity

durability

properties of transactions

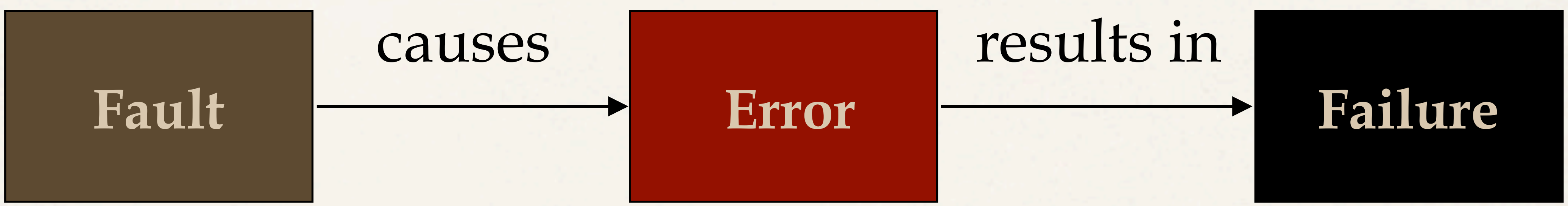
Fundamental Definitions

- 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*.

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.

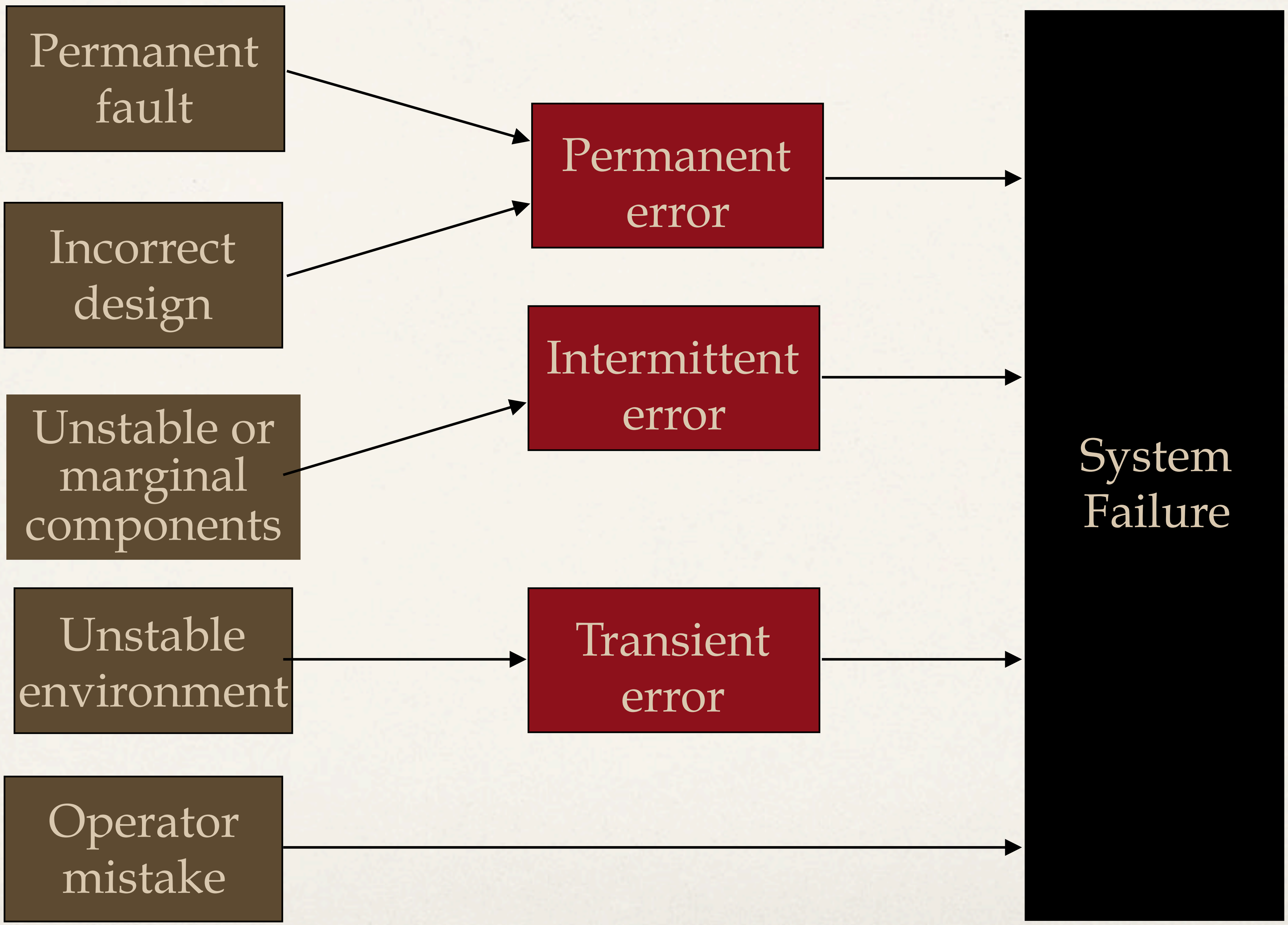
Faults to Failures



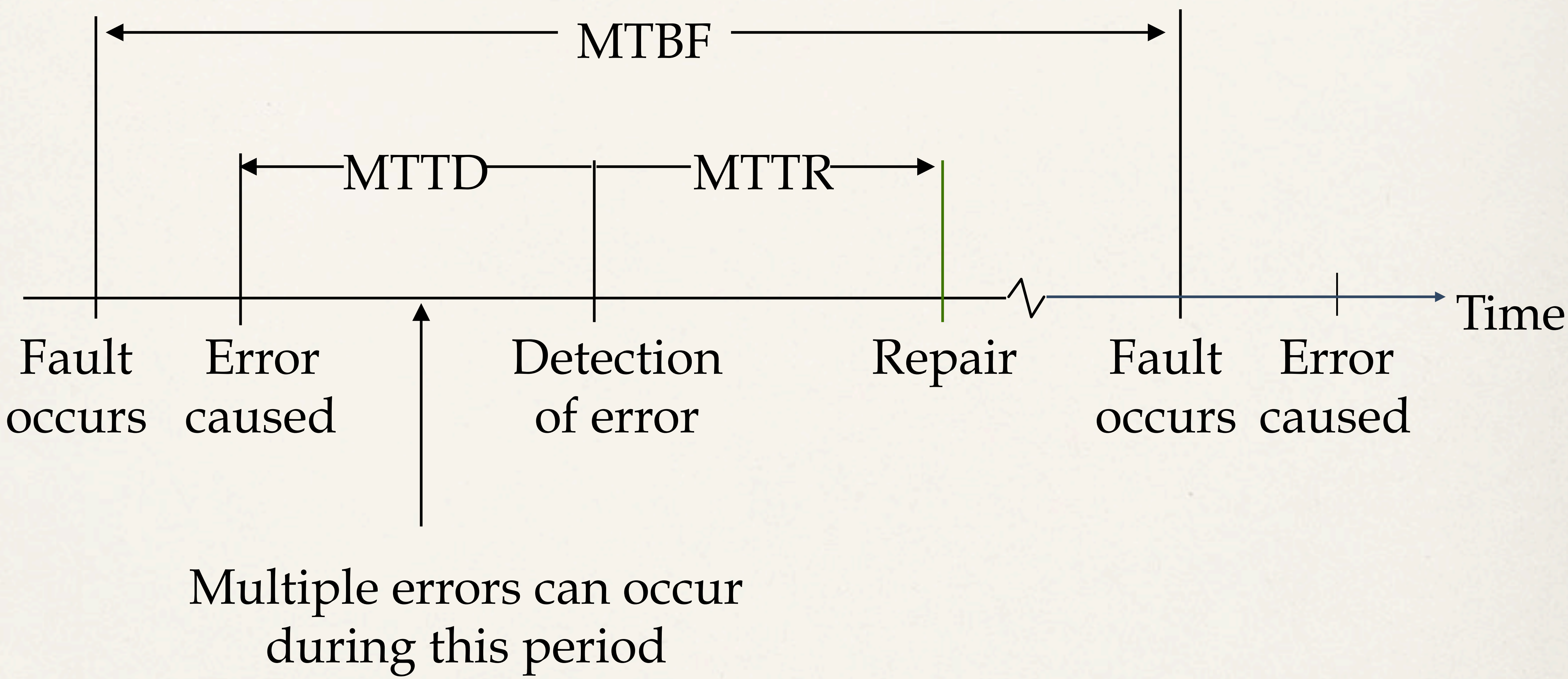
Types of Faults

- 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
MTTD
MTTR
Time
Fault occurs
Error caused
Detection of error
Repair
Fault occurs
Error caused
Multiple errors can occur during this period

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

$$\Pr(k \text{ failures in time } [0,t]) = \frac{e^{-m(t)}[m(t)]^k}{k!}$$

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

$z(x)$ is known as the hazard function which gives the time-dependent failure rate of the component

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 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

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

MTTR

Mean time to repair

Availability

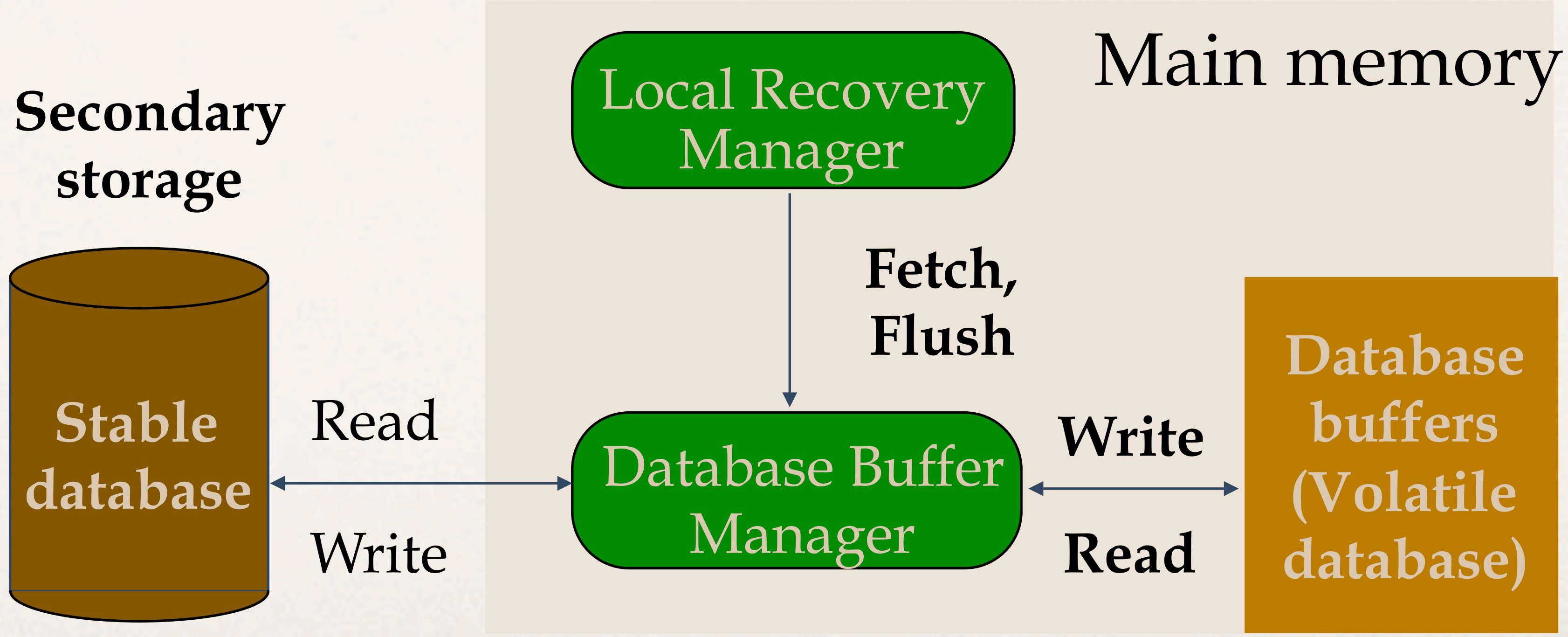
$$\frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

Types of Failures

- Transaction failures
  - Transaction aborts (unilaterally or due to deadlock)
  - Avg. 3% of transactions abort abnormally
- System (site) failures
  - Failure of processor, 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.



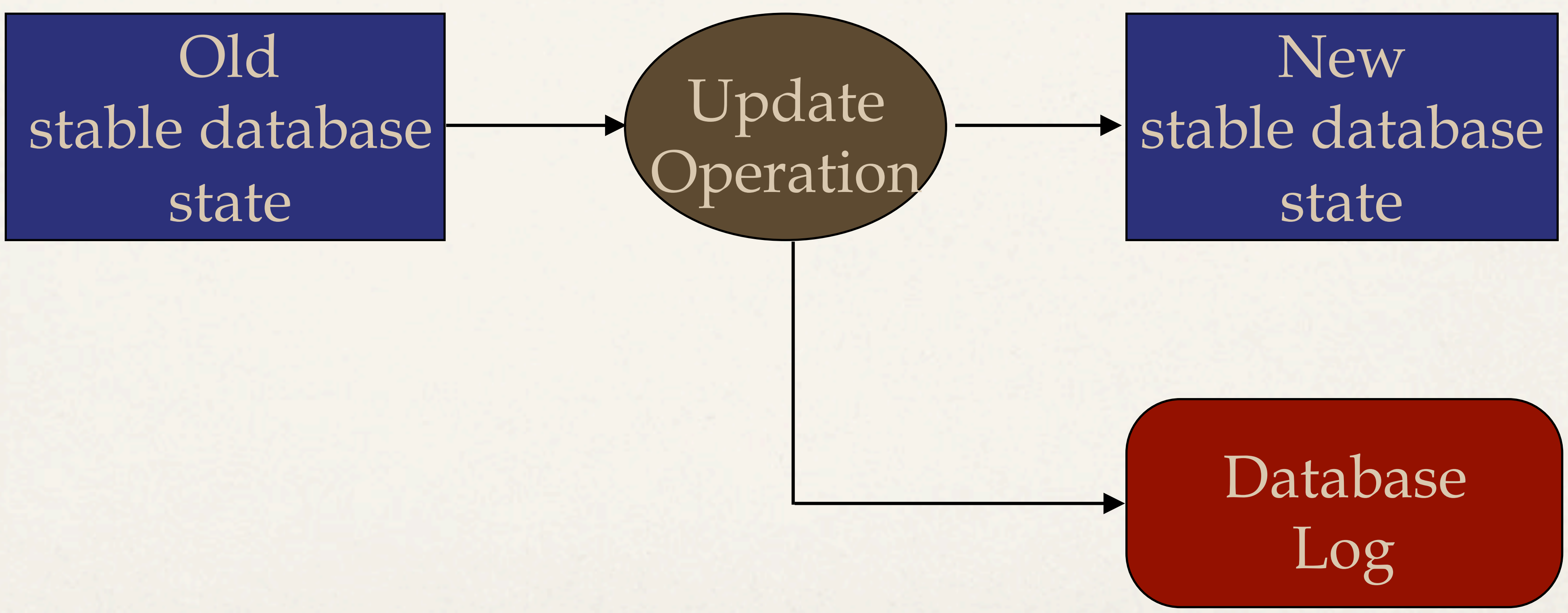
Update Strategies

- 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

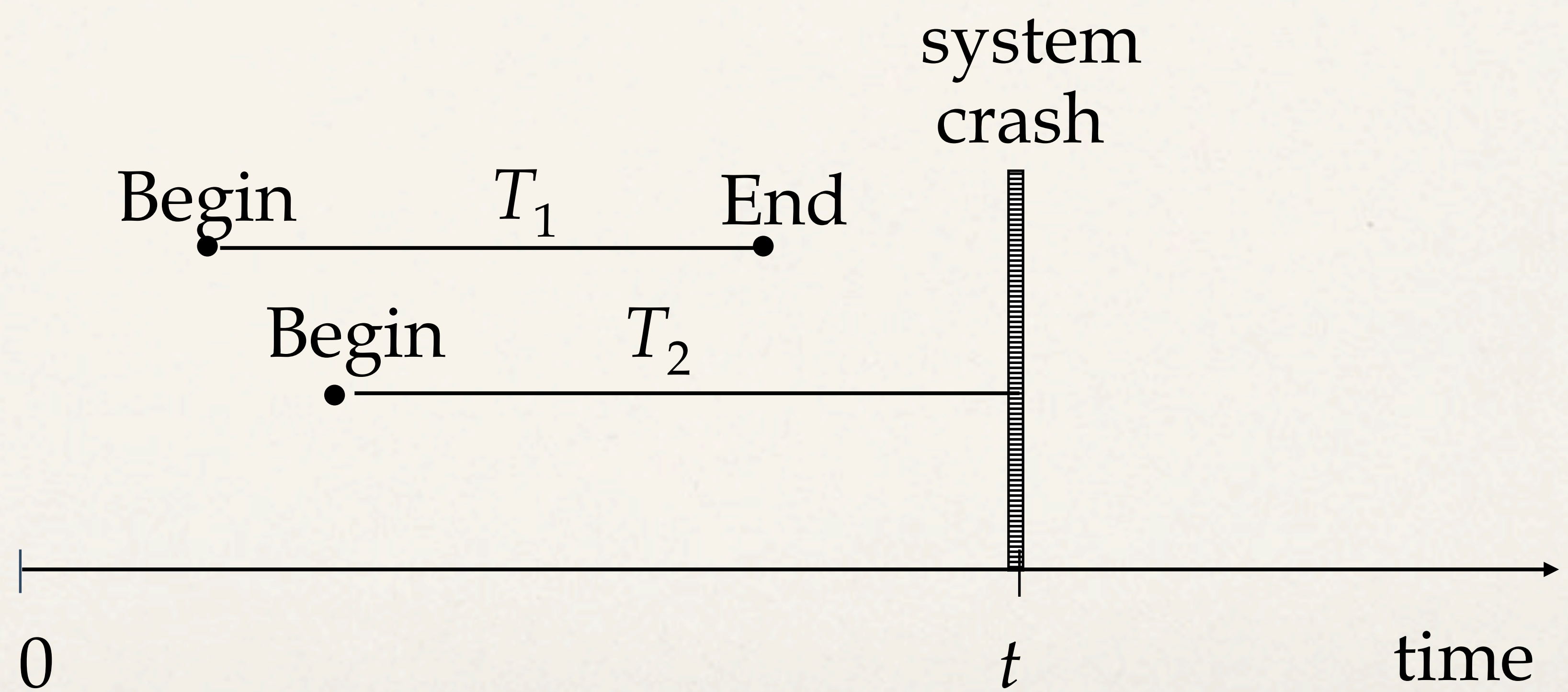
- ➡ transaction identifier
- ➡ type of operation (action)
- ➡ items accessed by the transaction to perform the action
- ➡ old value (state) of item (before image)
- ➡ new value (state) of item (after image)

    …

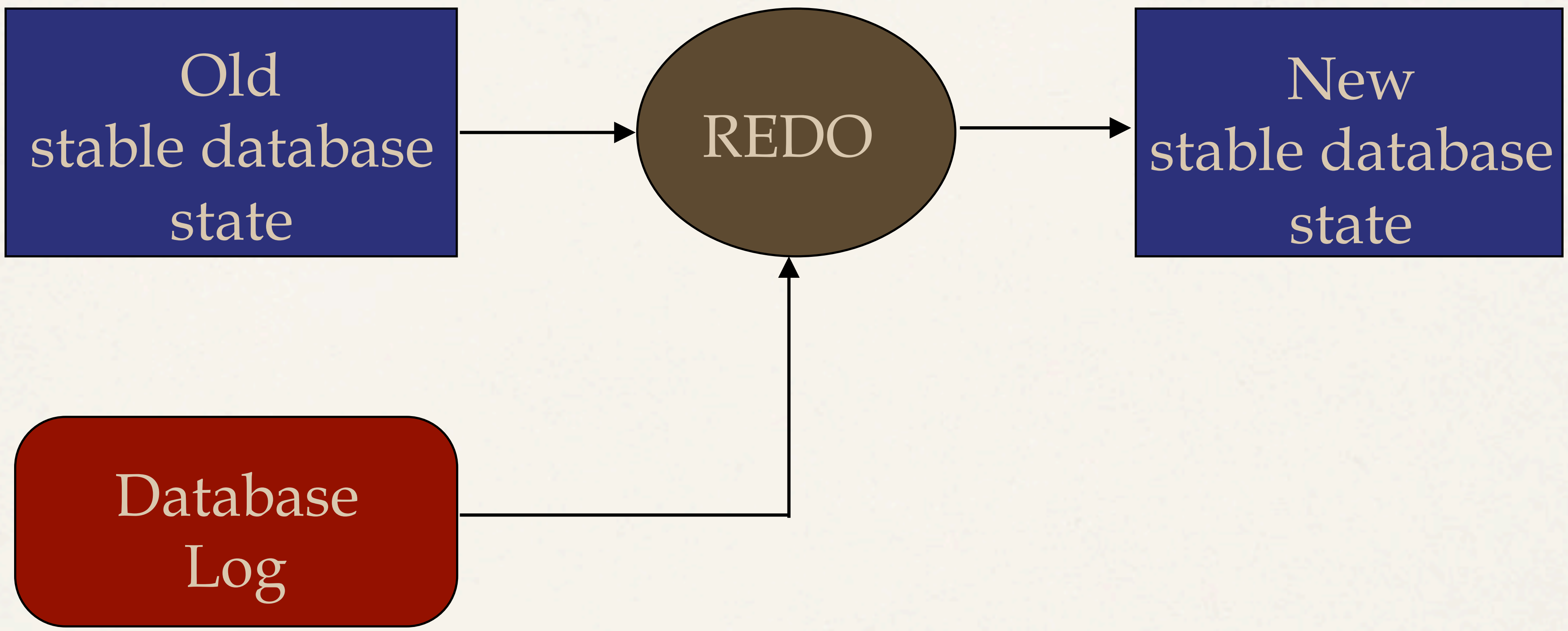
Why Logging?

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)

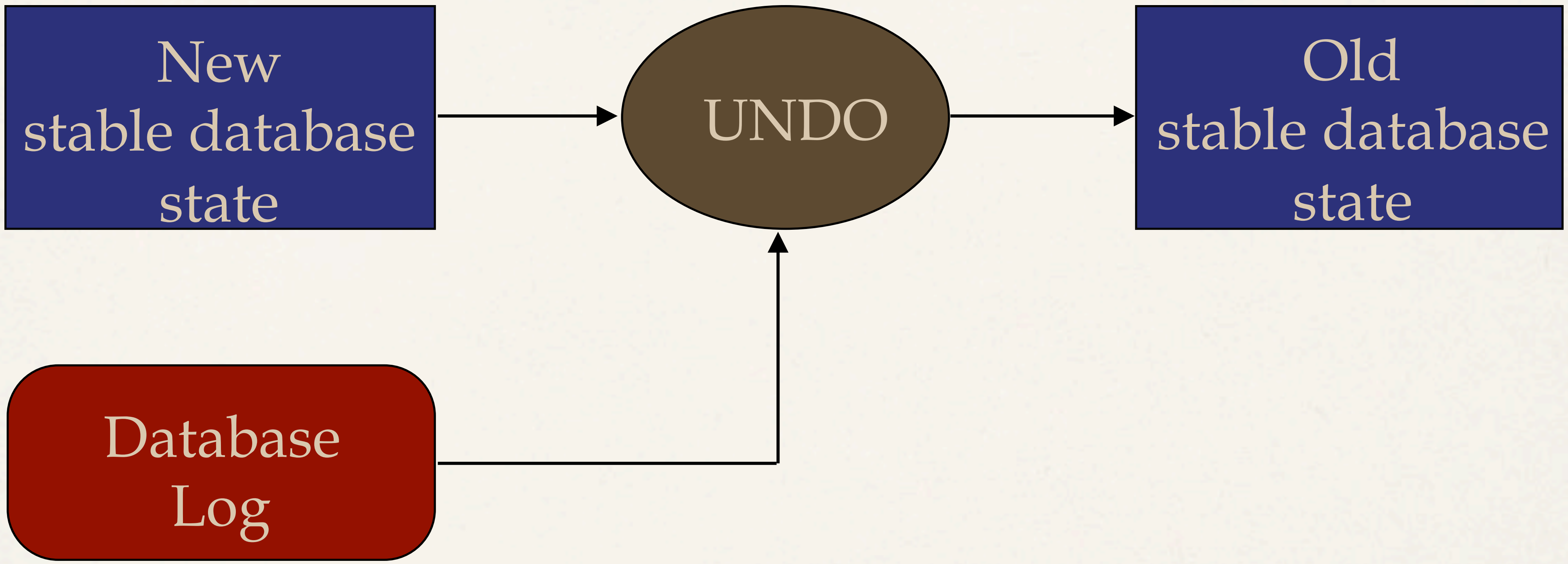


REDO Protocol



- 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)

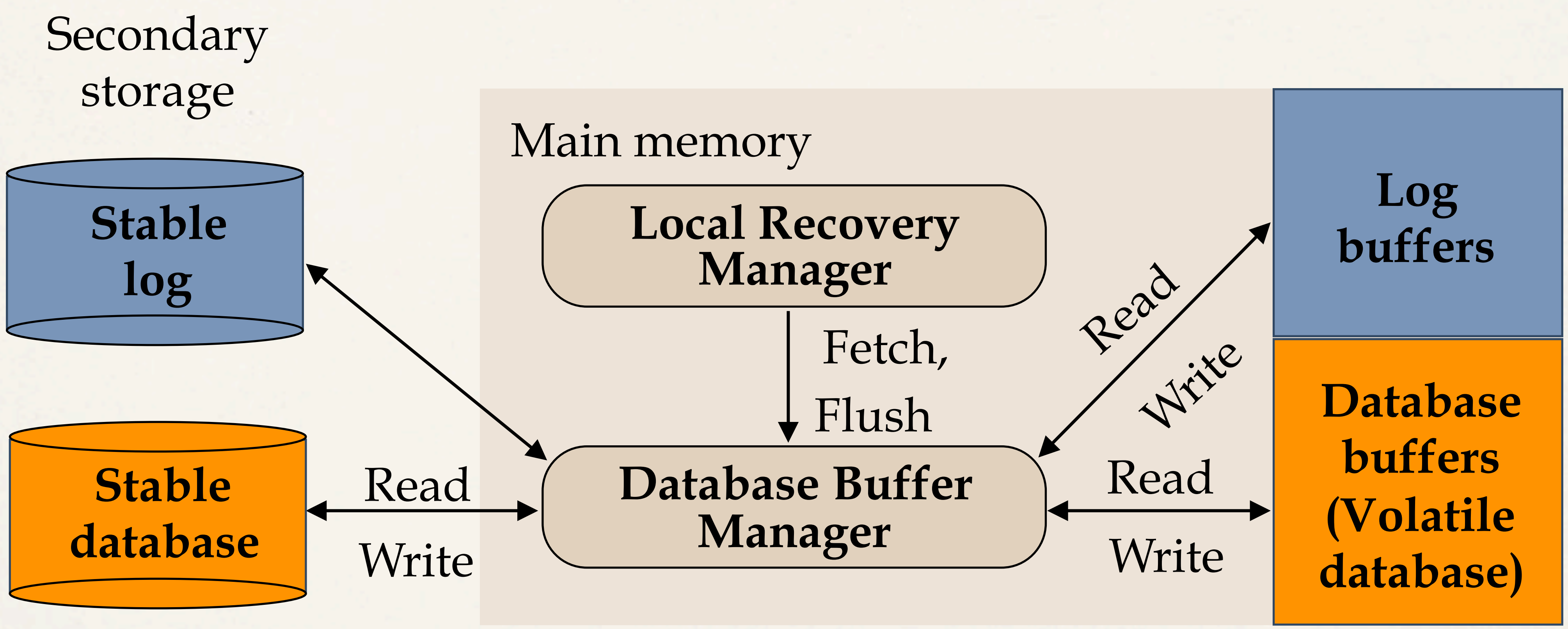
  We can recover (undo) by restoring *P* to its old state by using the log
- Unfortunate case
  - ➡ System writes *P* in stable database
  - ➡ SYSTEM FAILURE OCCURS!... (before stable log is updated)

  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 committed, some of its actions might have to be redone. Need the after images (*redo portion* of the log).
- WAL protocol :
  1. Before a stable database is updated, the undo portion of the log should be written to the stable log
  2. 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 part FR
    - a differential file consisting of insertions part DF^+ and deletions part DF^-
    - Thus, $F = (FR \cup DF^+) - DF^-$
  - Updates treated as delete old value, insert new value

Execution of Commands

Commands to consider:

- begin_transaction
- read
- write
- commit
- abort
- recover

Independent of execution strategy for LRM

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
  - no-fix/flush
  - fix/no-flush
  - fix/flush

No-Fix/No-Flush

- 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

- 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

- Abort
  - ➡ None of the updated pages have been written into stable database
  - ➡ Release the `fixed` 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 `fixed`
- Recover
  - ➡ Perform partial redo
  - ➡ No need to perform global undo

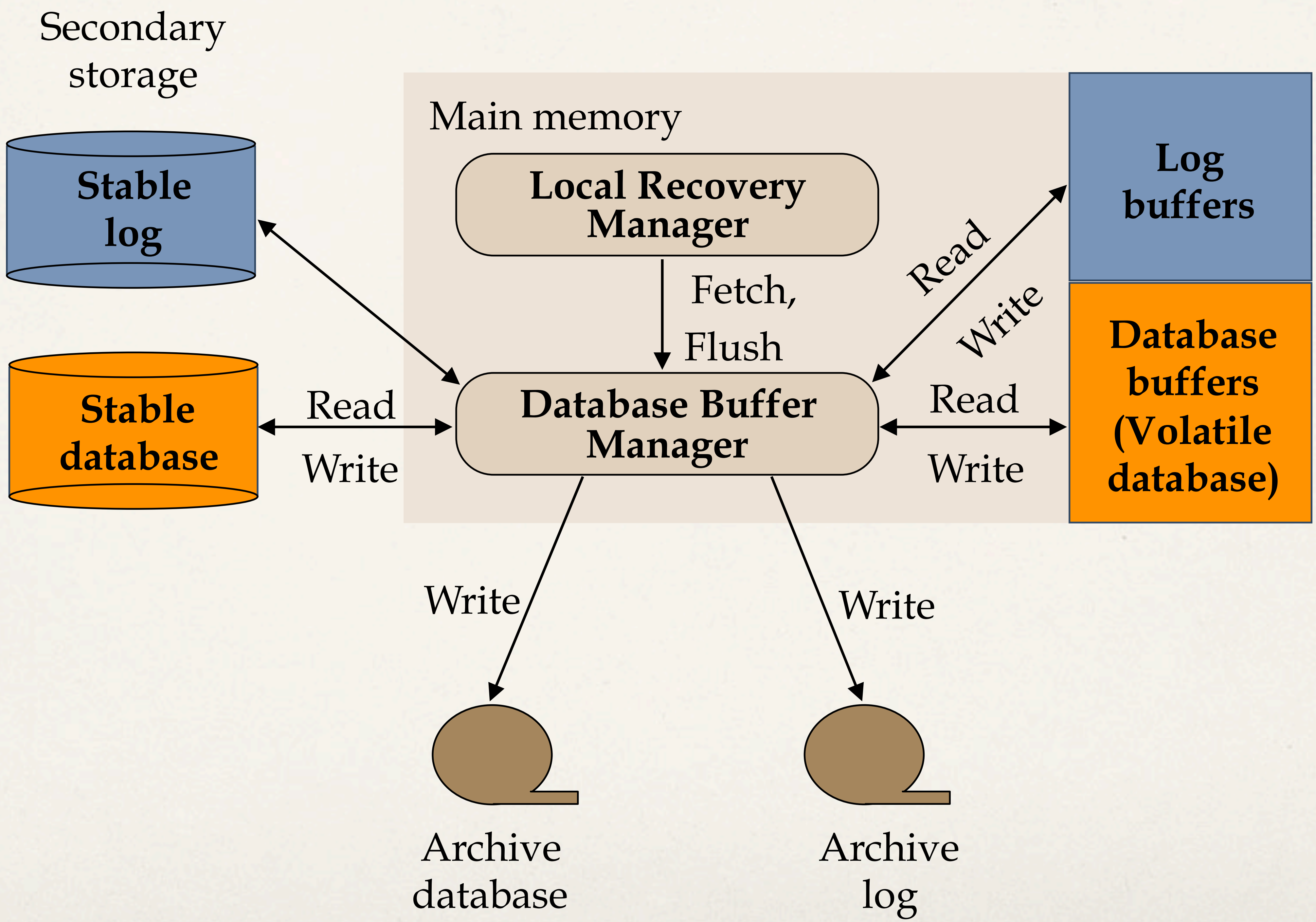
Fix/Flush

- Abort
  - ➡ None of the updated pages have been written into stable database
  - ➡ Release the `fixed` 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 `fixed`
  - ➡ 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:
  1. Write a begin_checkpoint record into the log
  2. Collect the checkpoint dat into the stable storage
  3. 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 ⇒ non-blocking termination

Two-Phase Commit (2PC)

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

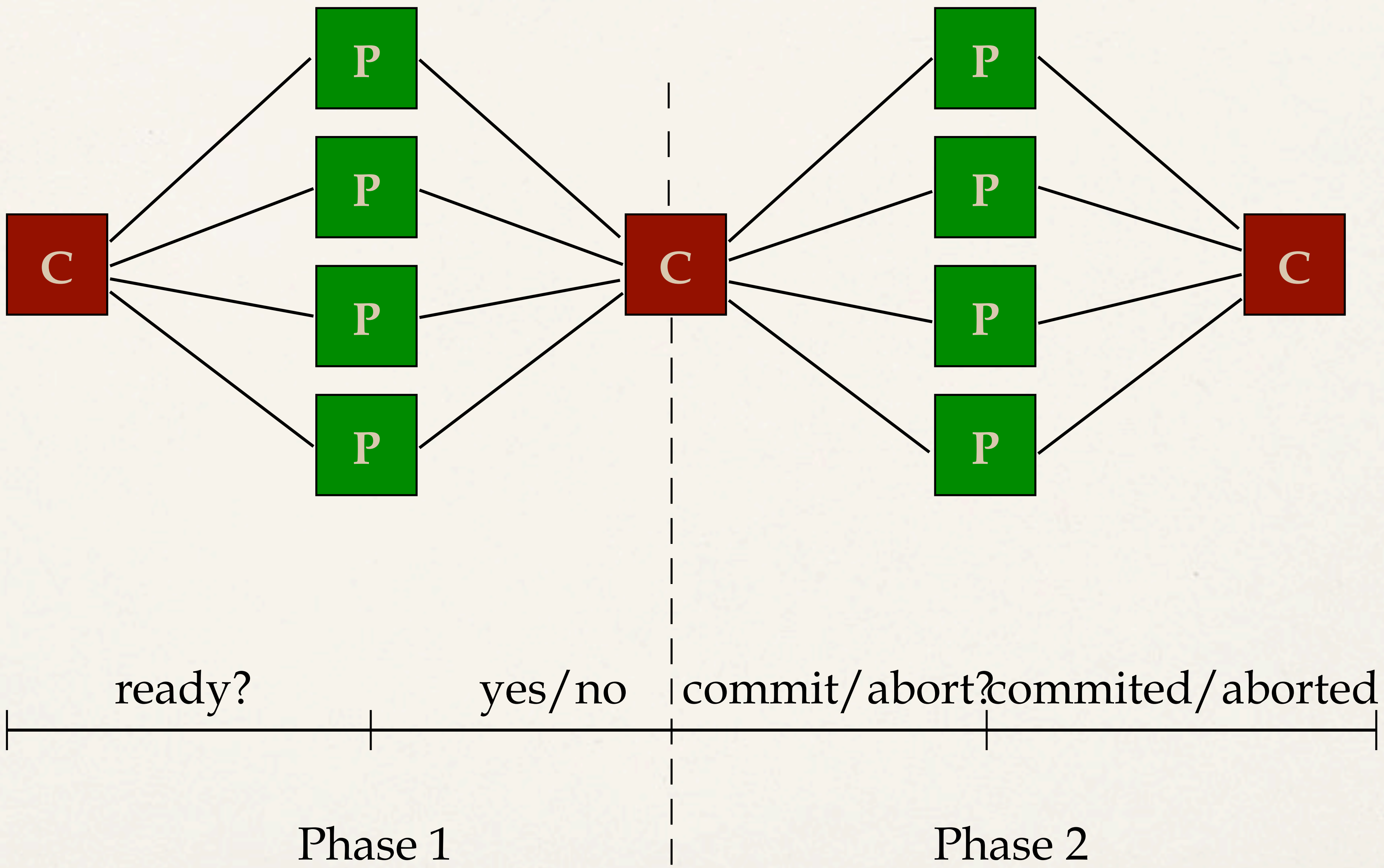
Phase 2 : Everybody writes the results into the database

- ➡ **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

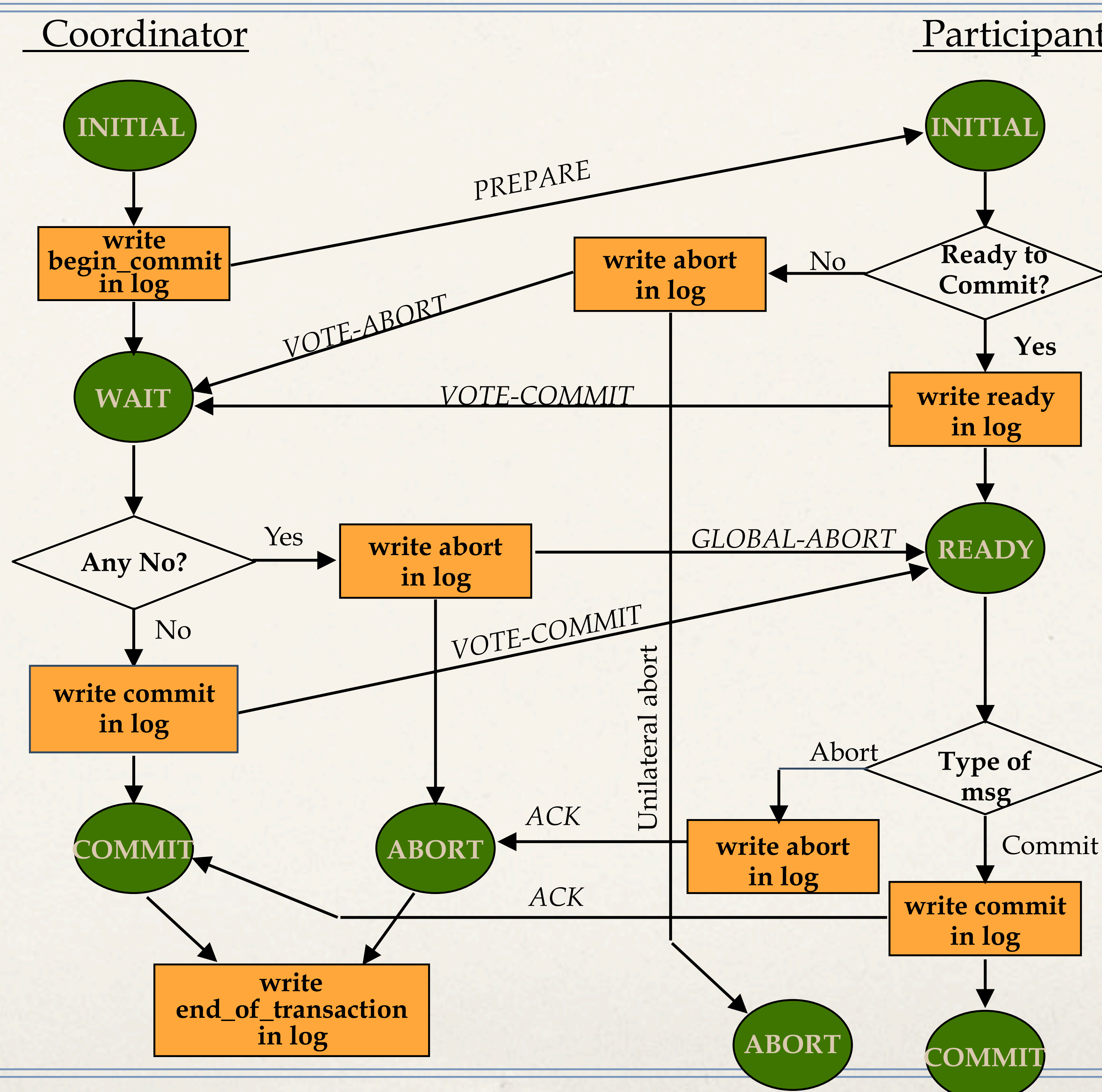
Global Commit Rule:

1. The coordinator aborts a transaction if and only if at least one participant votes to abort it.
2. The coordinator commits a transaction if and only if all of the participants vote to commit it.

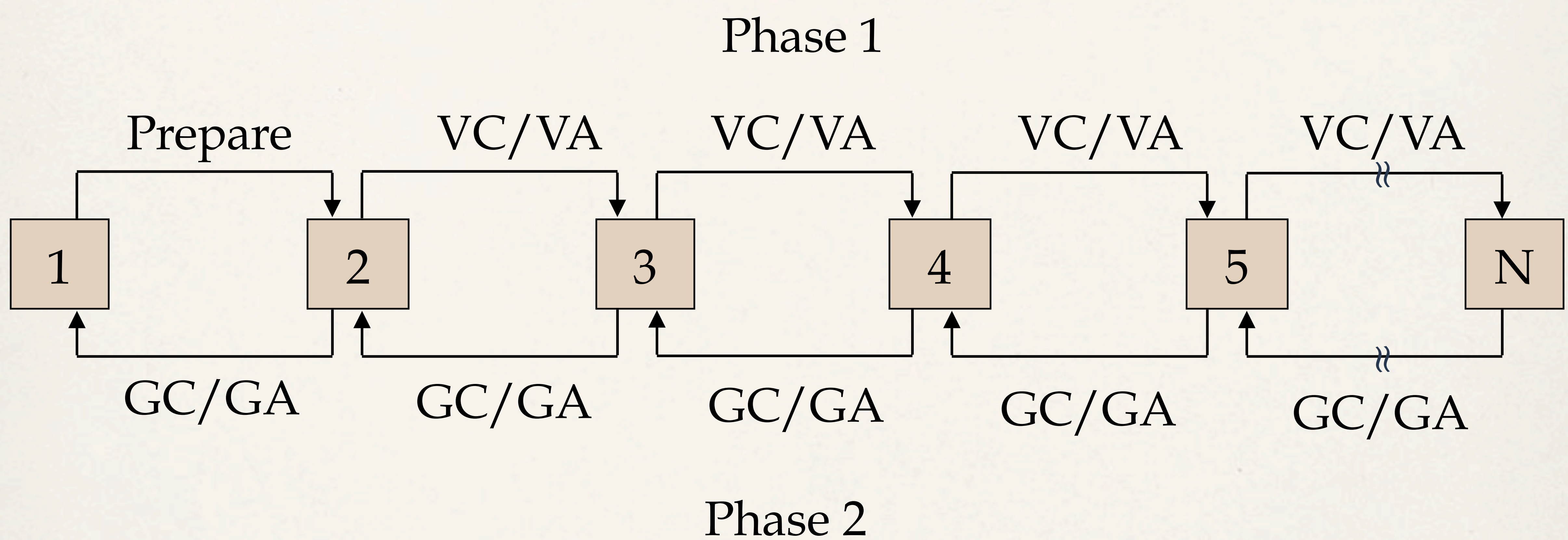
Centralized 2PC



2PC Protocol Actions

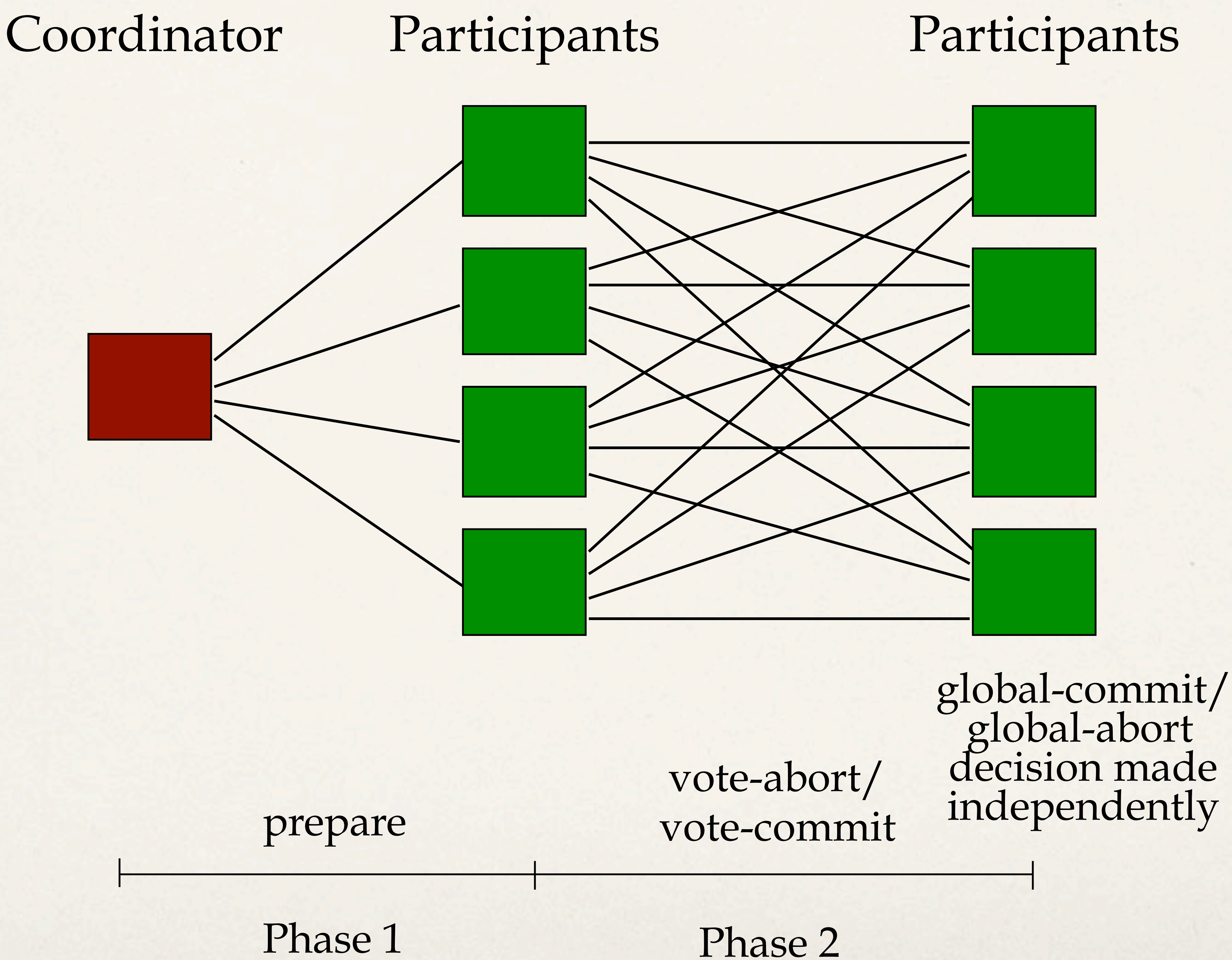


Linear 2PC

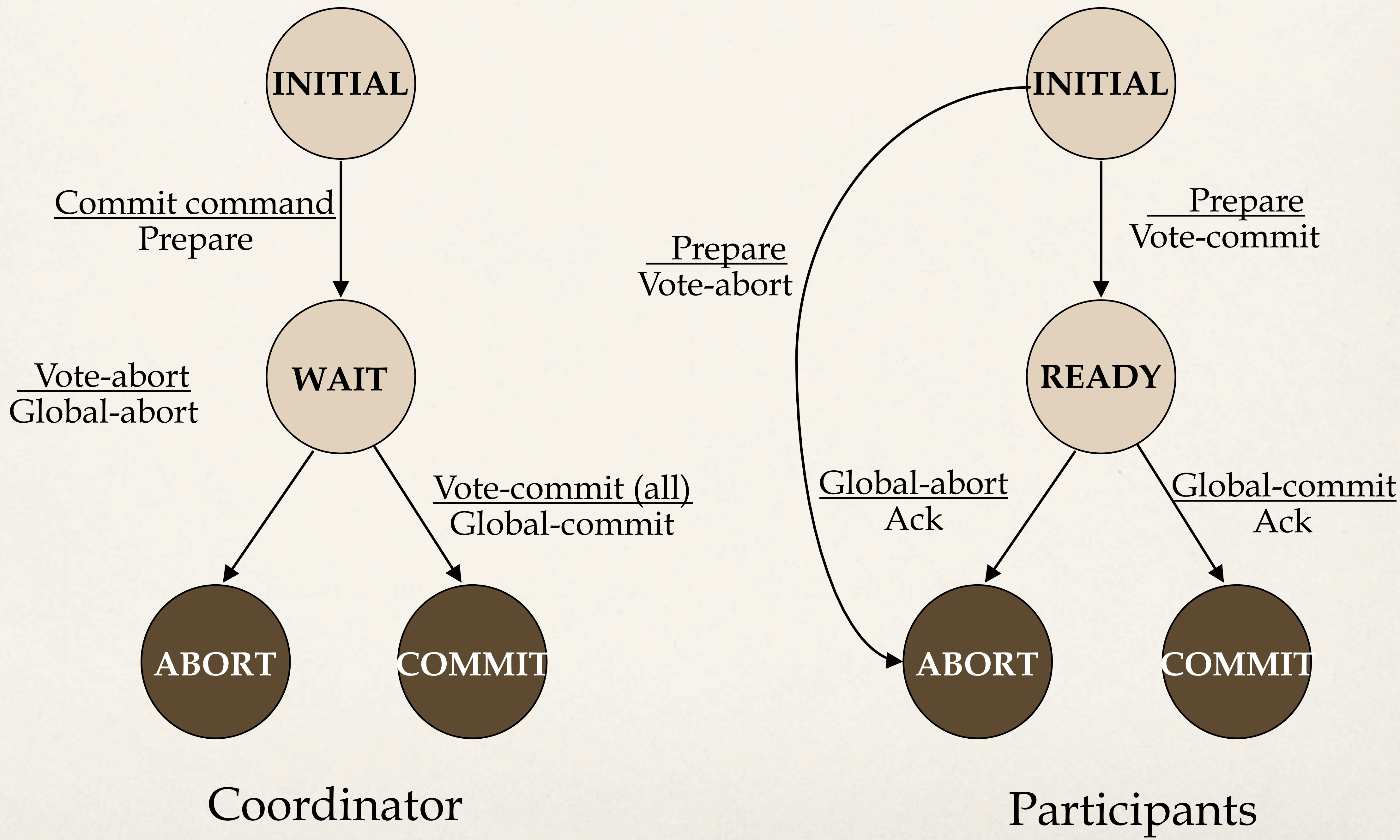


VC: Vote-Commit, VA: Vote-Abort, GC: Global-commit, GA: Global-abort

Distributed 2PC

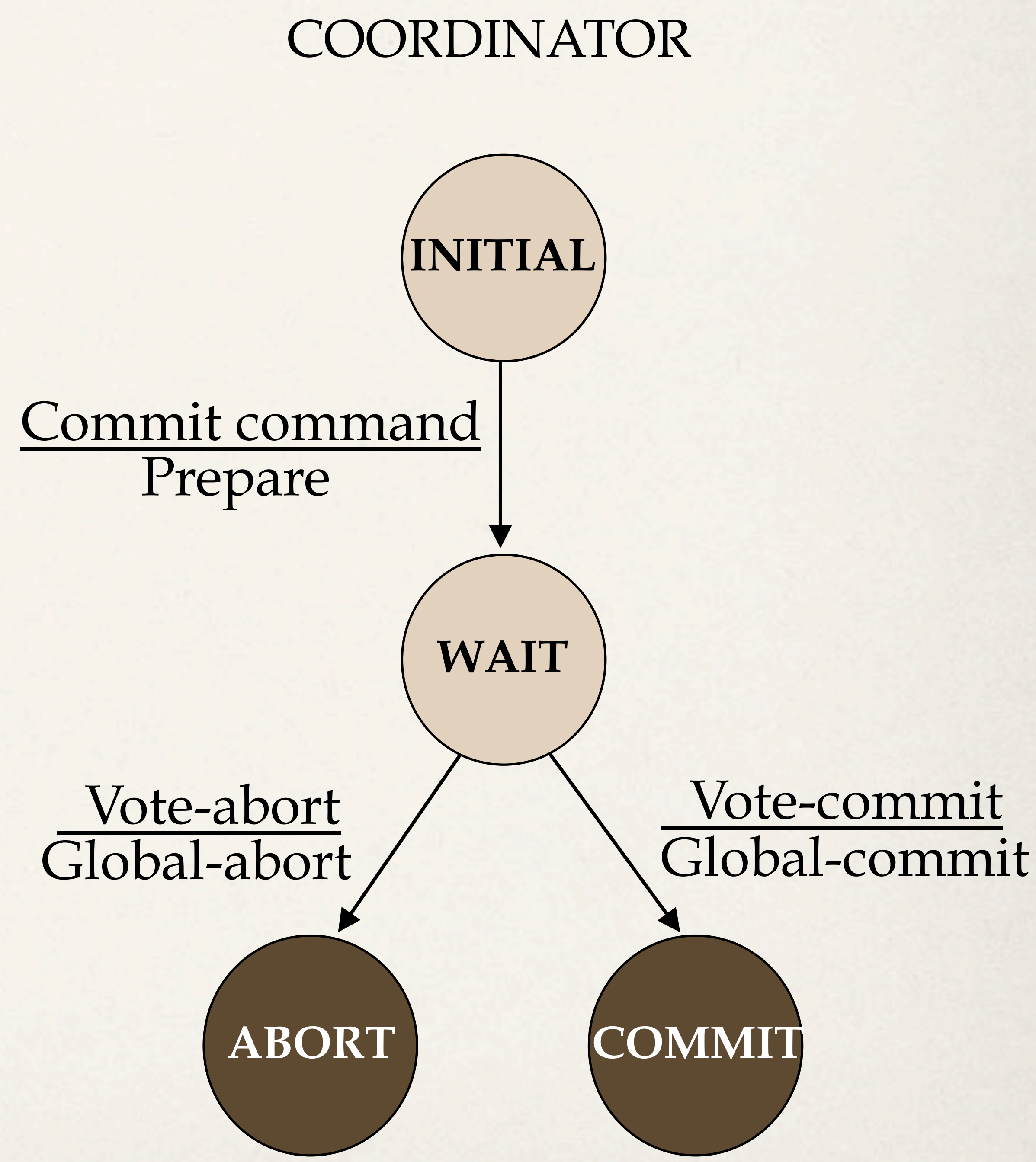


State Transitions in 2PC



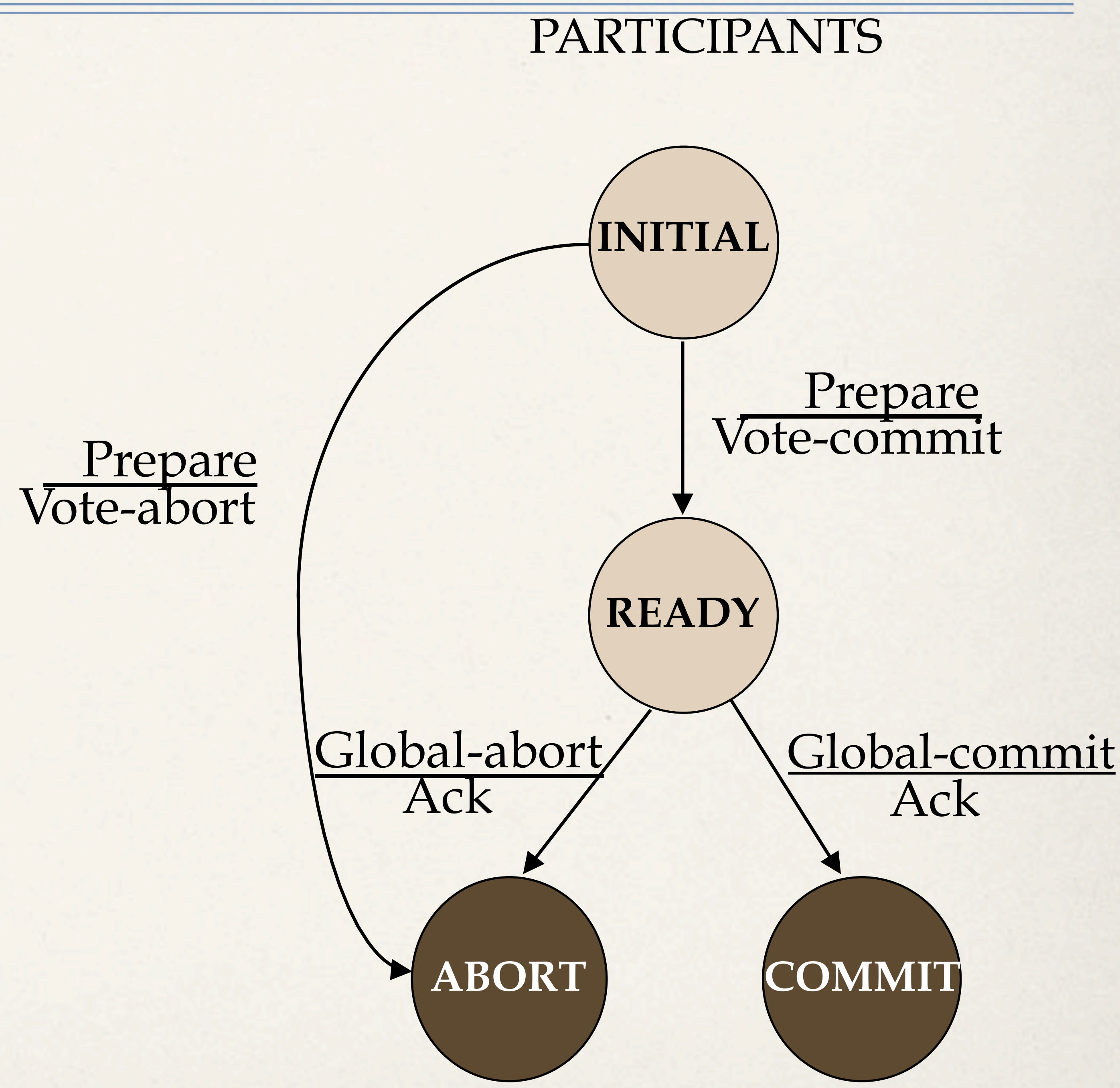
Site Failures - 2PC Termination

- 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



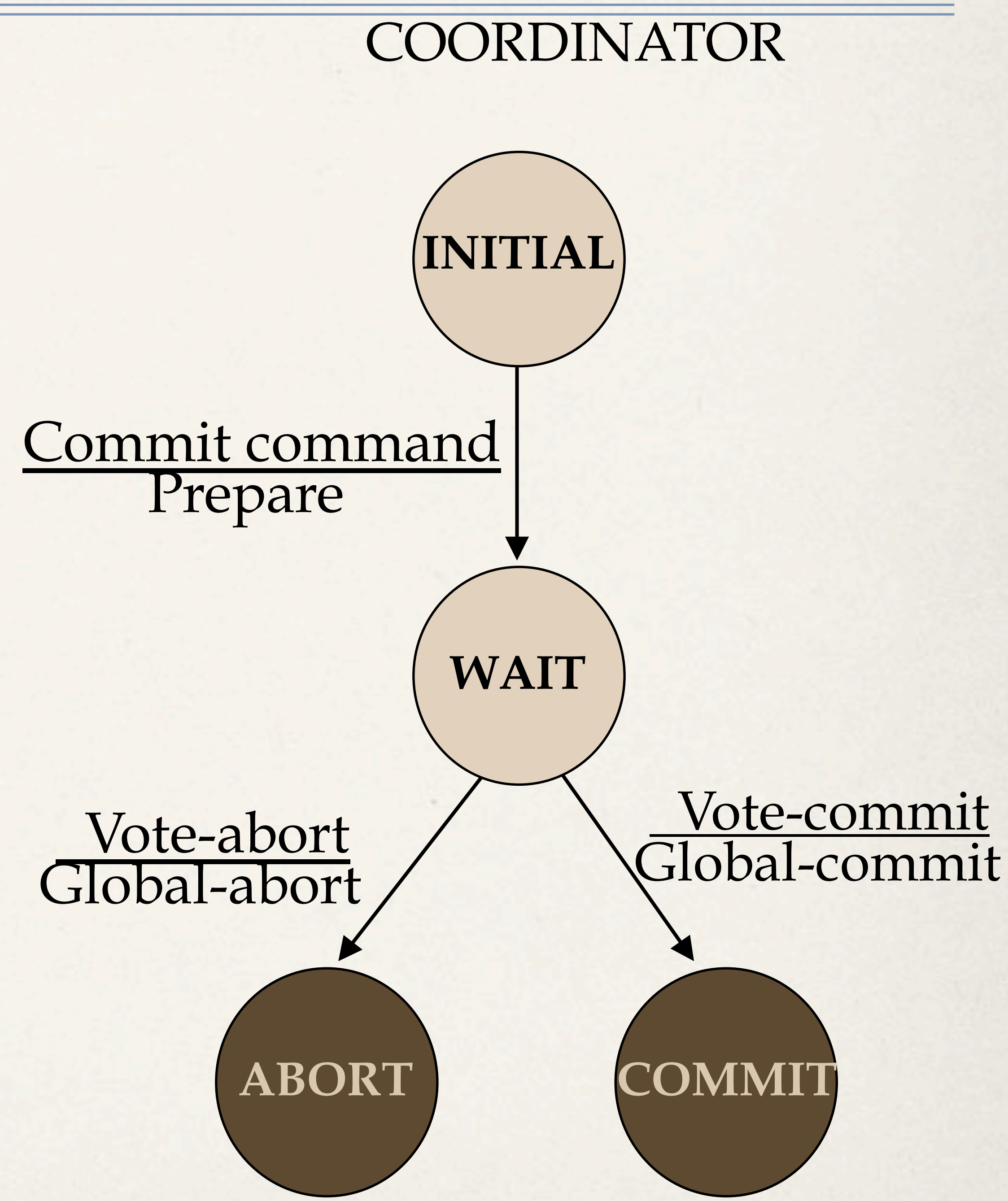
Site Failures - 2PC Termination

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



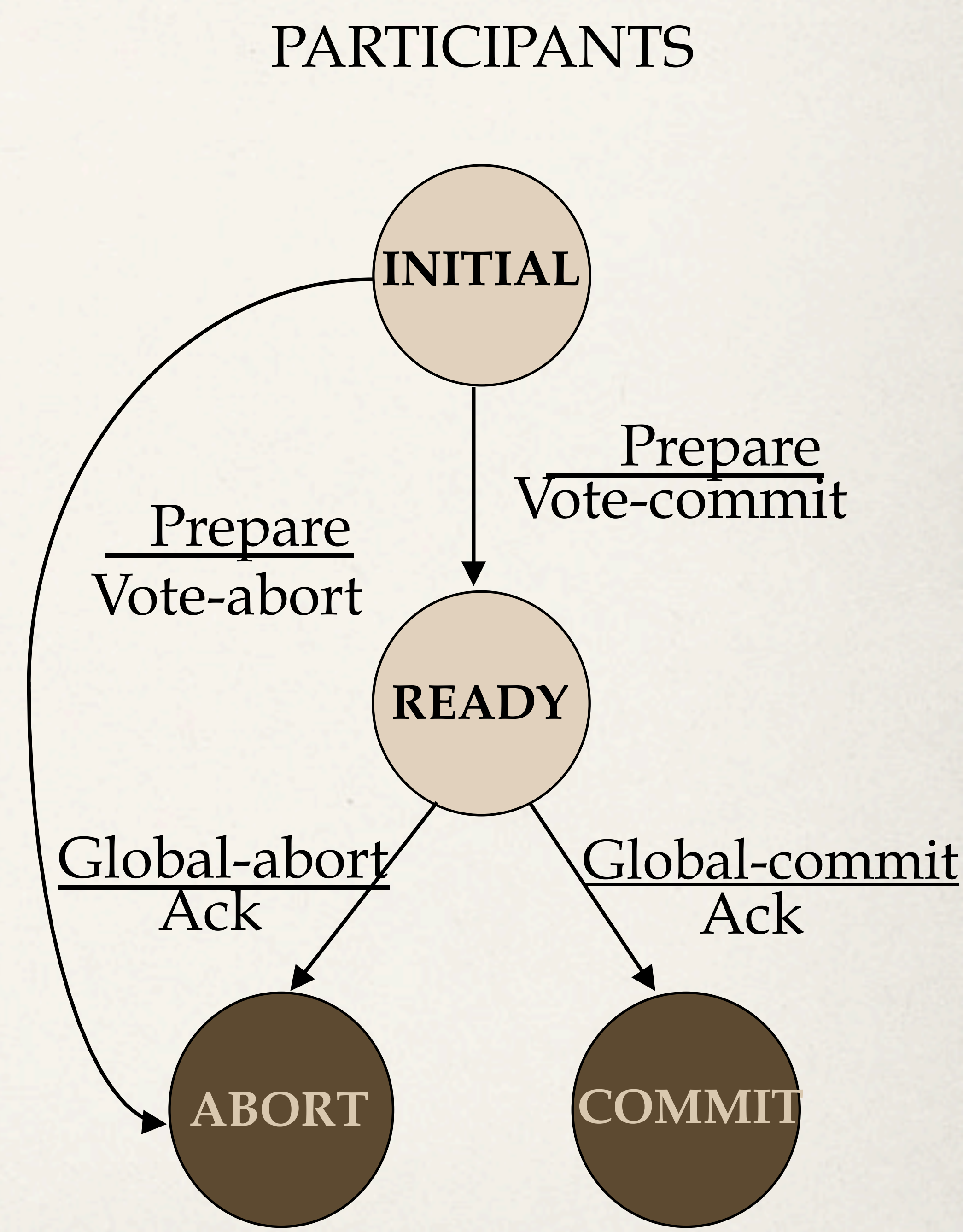
Site Failures - 2PC Recovery

- 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



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 sending “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

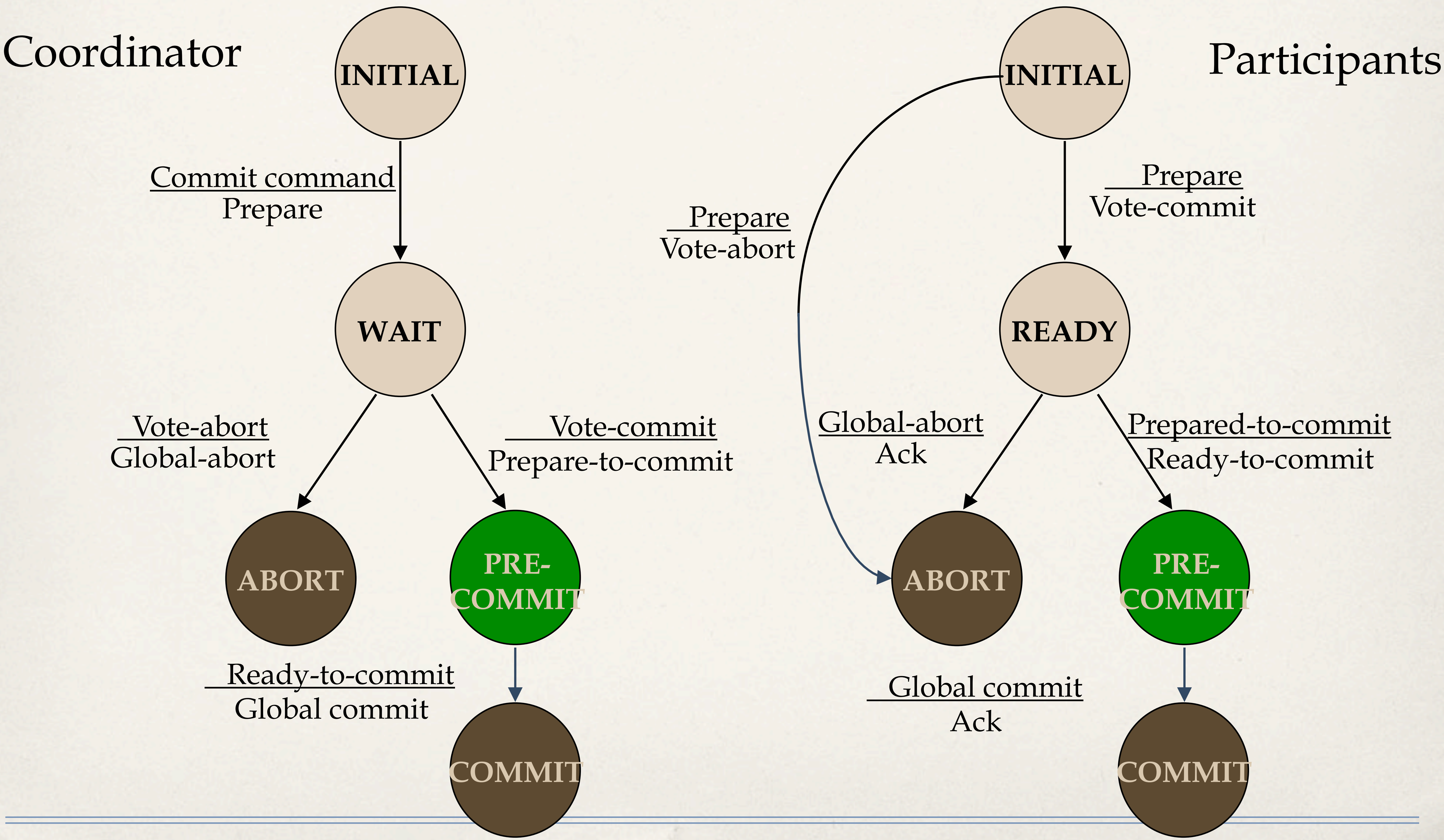
Problem With 2PC

- Blocking
  - ➡ Ready implies that the participant waits for the coordinator
  - ➡ If coordinator fails, site is blocked until recovery
  - ➡ Blocking reduces availability
- 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 protocols – 3PC

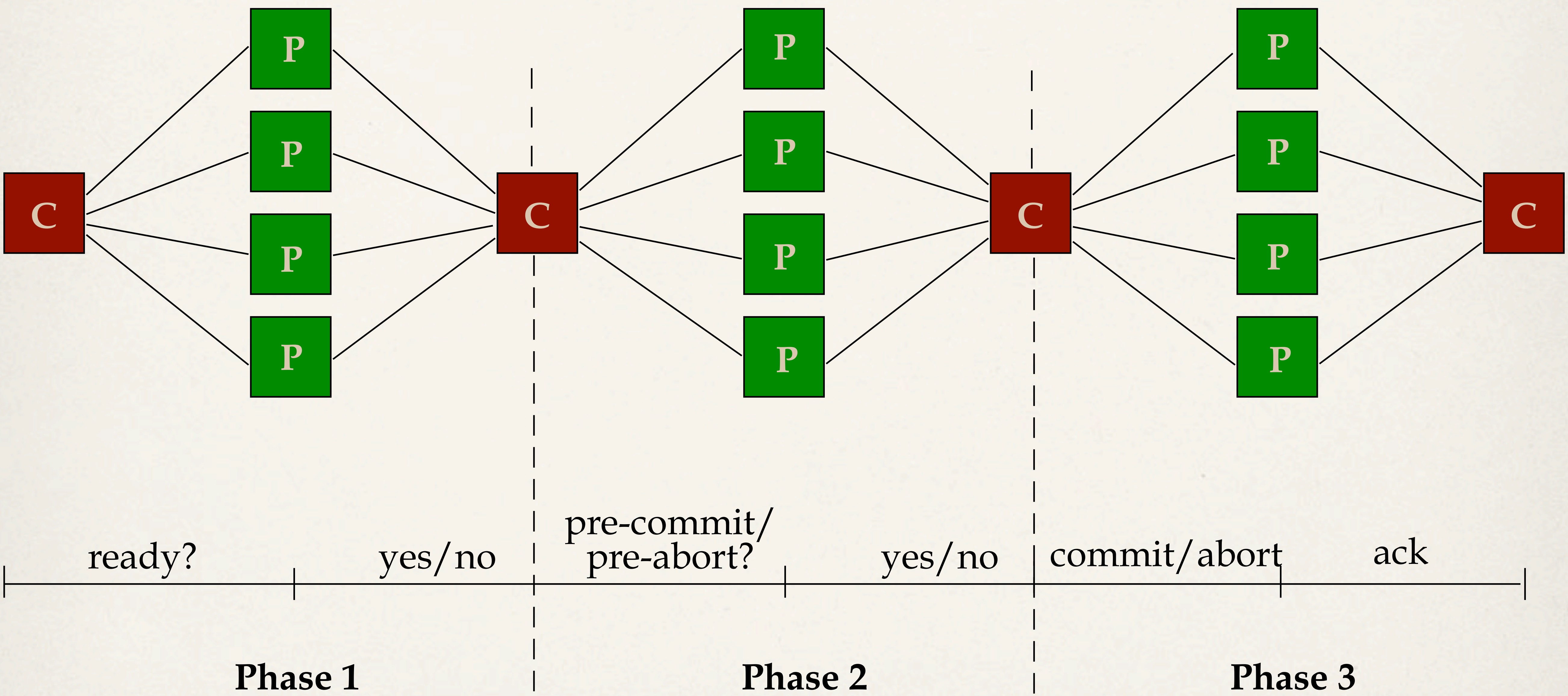
Three-Phase Commit

- 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 stat to another with a single state transition
- Committable: all sites have voted to commit a transaction
  - e.g.: COMMIT state

State Transitions in 3PC

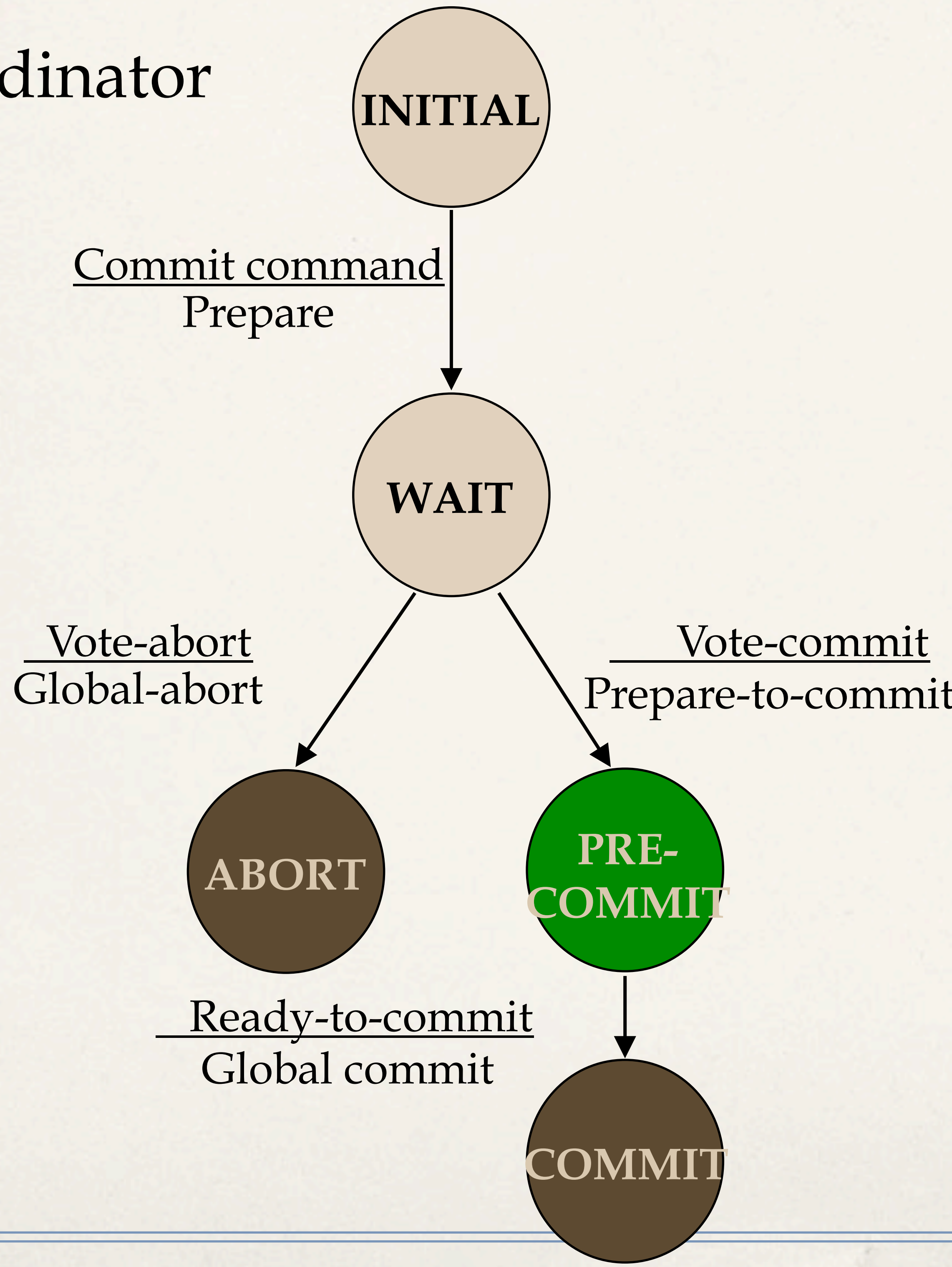


Communication Structure



Site Failures – 3PC Termination

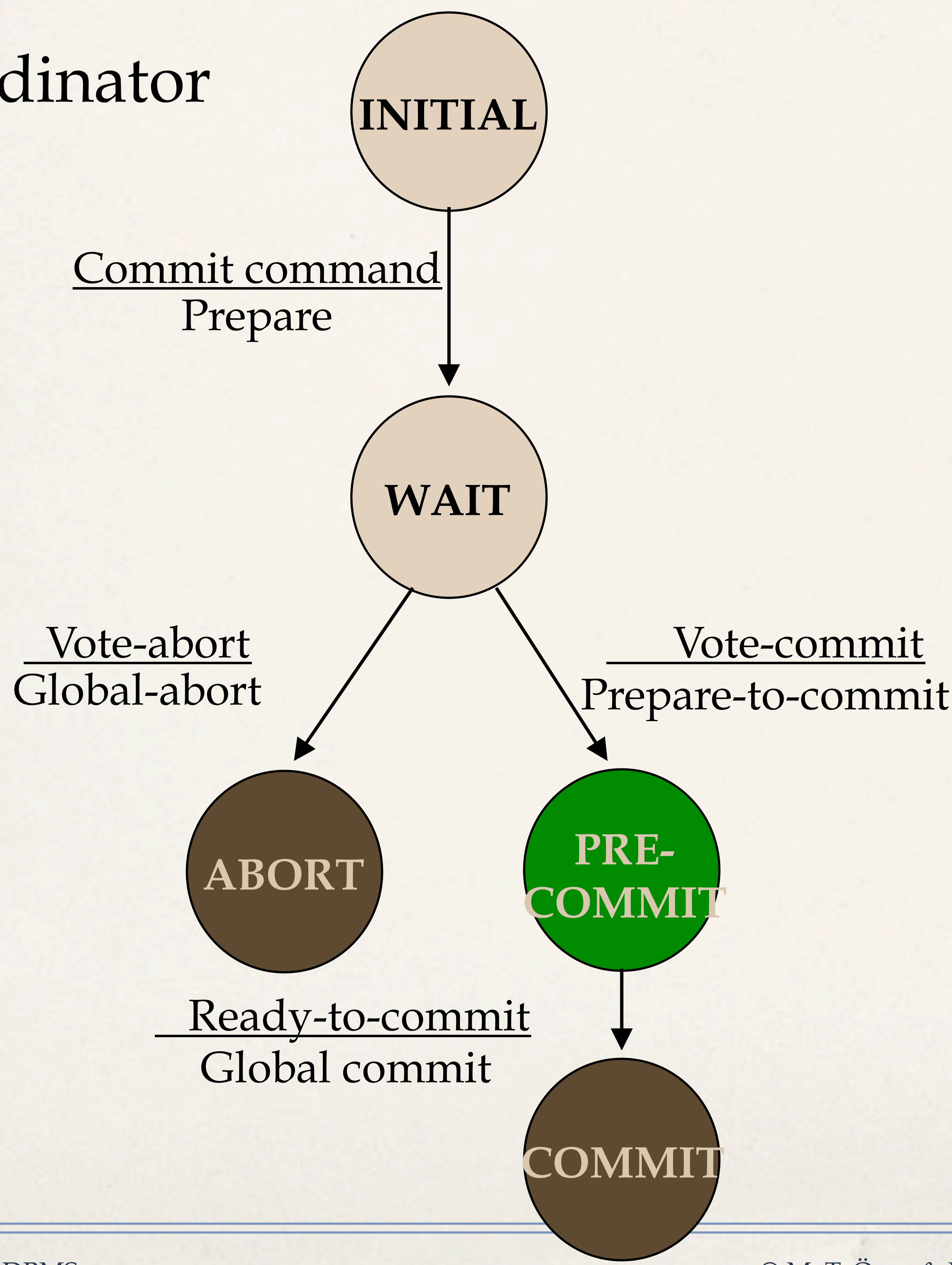
Coordinator



- Timeout in INITIAL
  - ➡ Who cares
- Timeout in WAIT
  - ➡ Unilaterally abort
- Timeout in PRECOMMIT
  - ➡ Participants may not be in PRE-COMMIT, 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

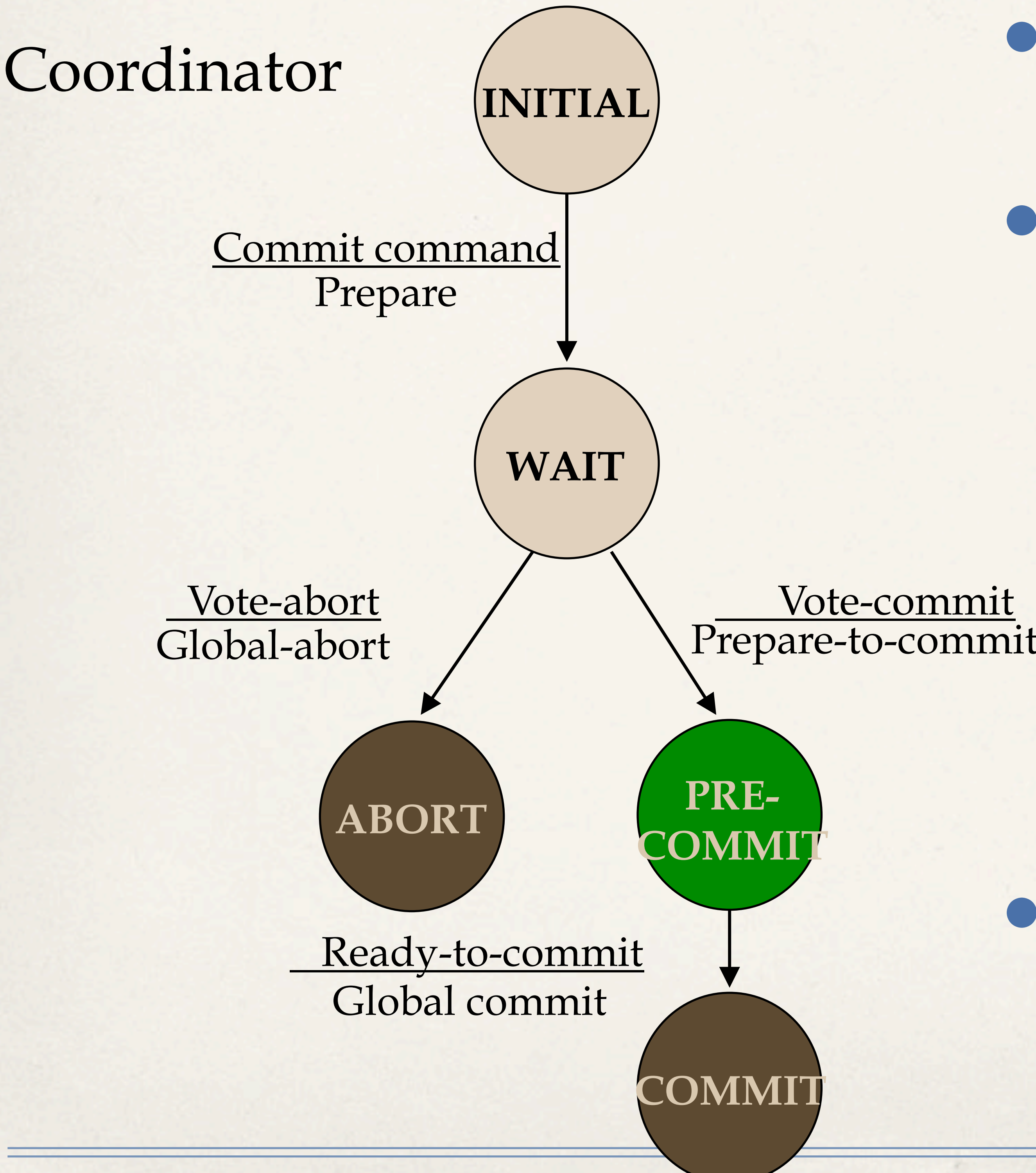
- Timeout in INITIAL
  - ➡ Coordinator must have failed in INITIAL state
  - ➡ Unilaterally abort
- Timeout in READY
  - ➡ Voted to commit, but does not know the coordinator's decision
  - ➡ Elect a new coordinator and terminate using a special protocol
- Timeout in PRECOMMIT
  - ➡ Handle it the same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

Termination Protocol Upon Coordinator Election

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

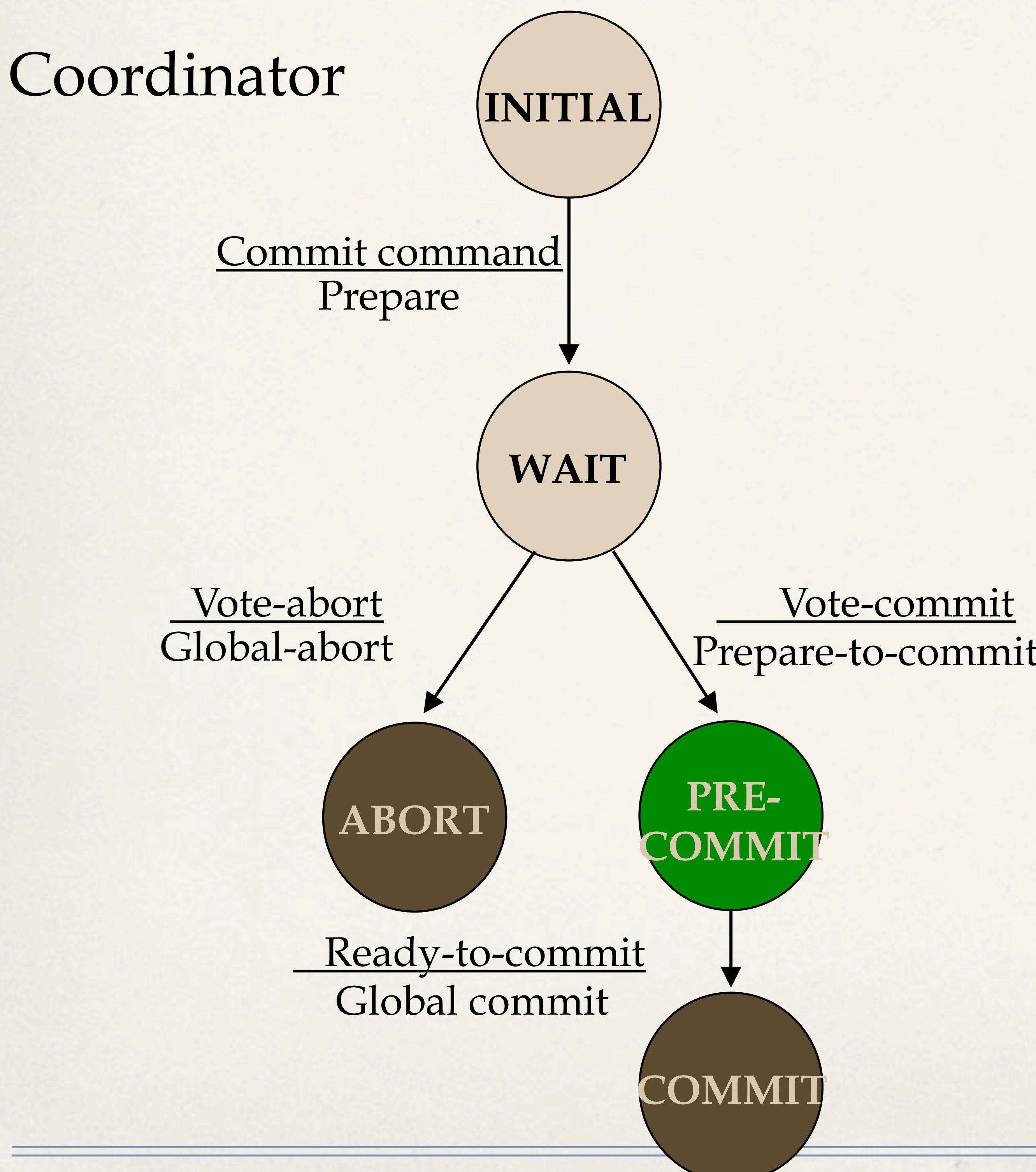
1. Coordinator sends its state to all of the participants asking them to assume its state.
2. Participants "back-up" and reply with appriate messages, except those in ABORT and COMMIT states. Those in these states respond with "Ack" but stay in their states.
3. Coordinator guides the participants towards 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



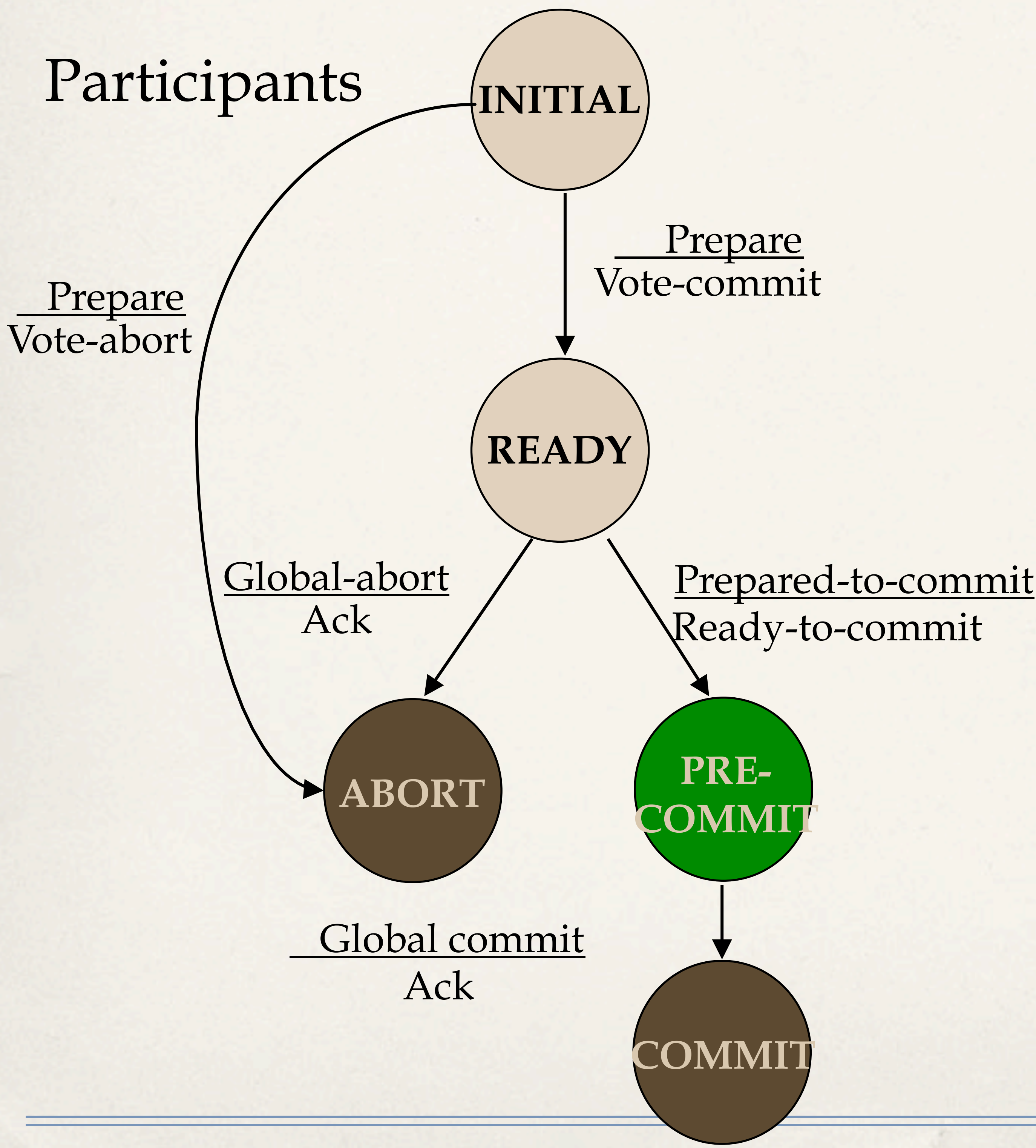
- Failure in INITIAL
  - ➡ start commit process upon recovery
- Failure in WAIT
  - ➡ the participants may have elected a new coordinator and terminated the transaction
  - ➡ the new coordinator could be in WAIT or ABORT states ➡ transaction aborted
  - ➡ ask around for the fate of the transaction
-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

Site Failures – 3PC Recovery



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

Site Failures – 3PC Recovery



- 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:
  - ➡ There exists no non-blocking protocol 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.

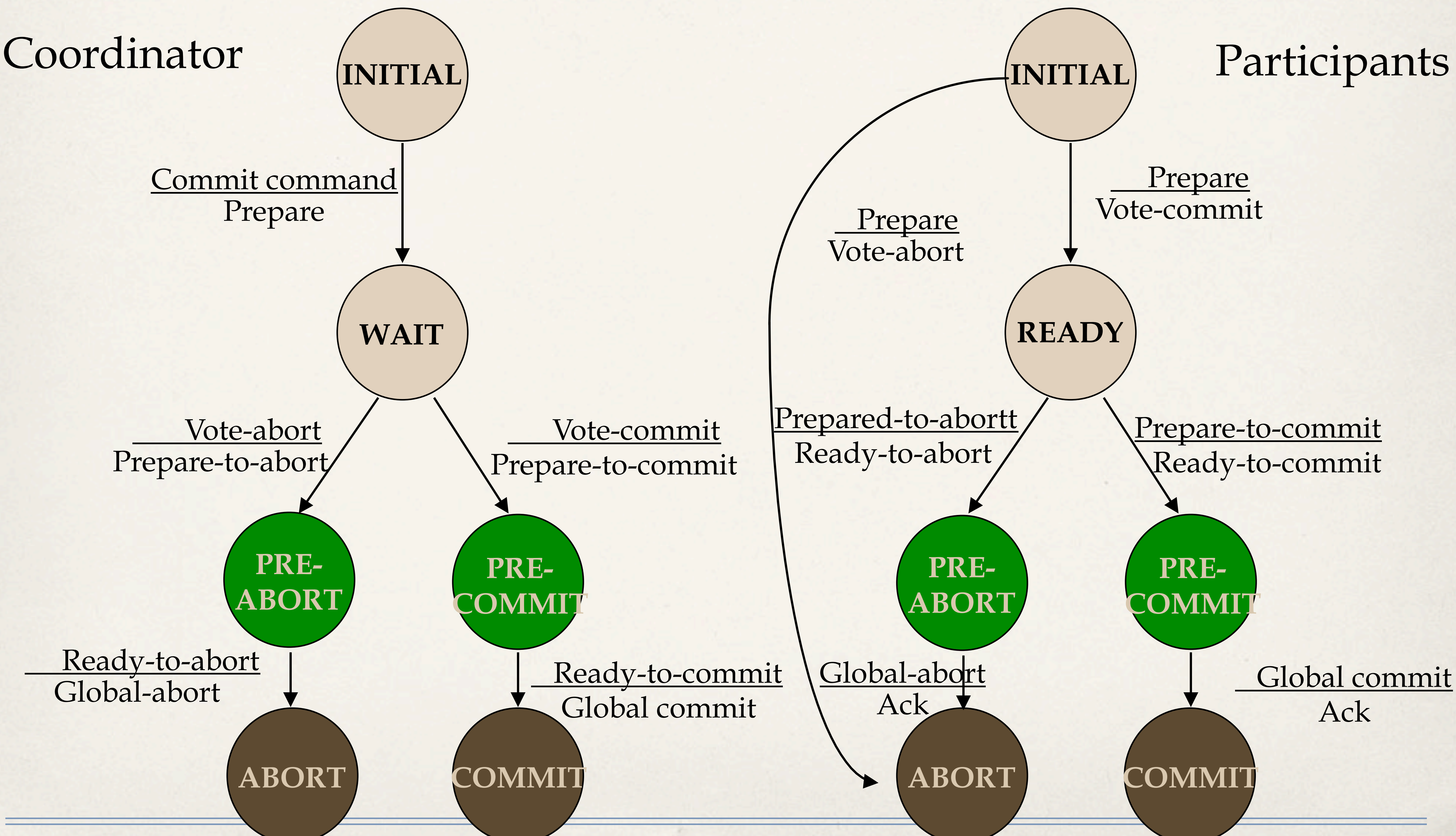
Independent Recovery Protocols for Network Partitioning

- 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)

Quorum Protocols

- 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



Use for Network Partitioning

- Before commit (i.e., moving from PRECOMMIT to COMMIT), coordinator receives commit quorum from participants. One partition may have the commit quorum.
- 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