Distributed Transaction Management

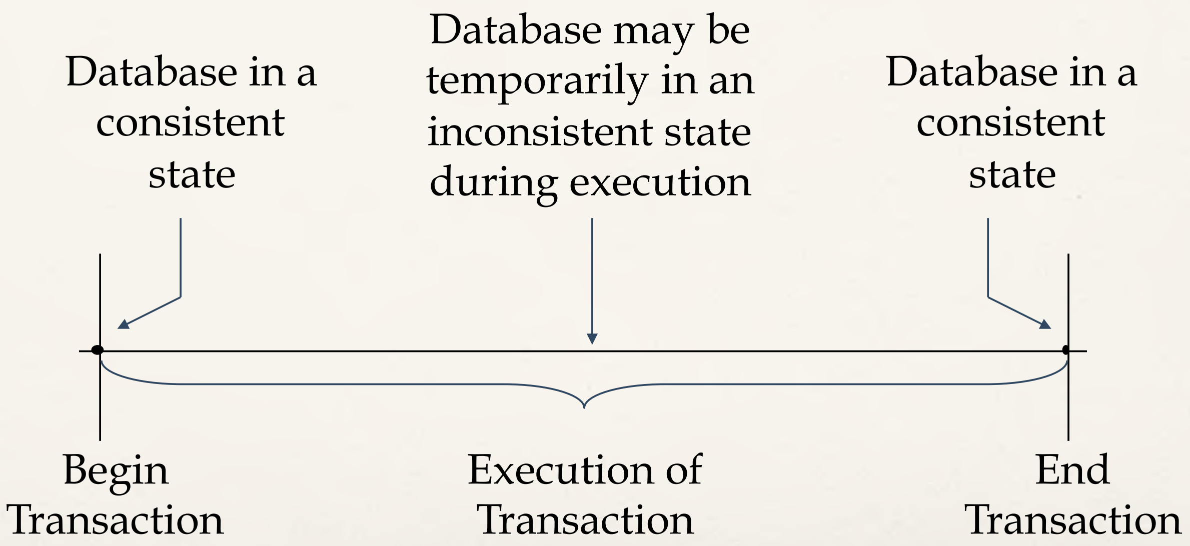
Material from:
Principles of Distributed Database Systems
Özsu, M. Tamer, **Valduriez**, Patrick, 3rd ed. 2011

+ Presented by C. Roncancio

Transaction

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

- ➡ concurrency transparency
- ➡ failure transparency


Database in a consistent state
Database may be temporarily in an inconsistent state during execution
Database in a consistent state
Begin Transaction
Execution of Transaction
End Transaction

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 ∪ WS

Principles of Transactions

ATOMICITY

- ➡ all or nothing

CONSISTENCY

- ➡ no violation of integrity constraints

ISOLATION

- ➡ concurrent changes invisible ⇒ 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

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.

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)
  - Structure
    - Flat (or simple) transactions
    - Nested transactions
    - Workflows

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)

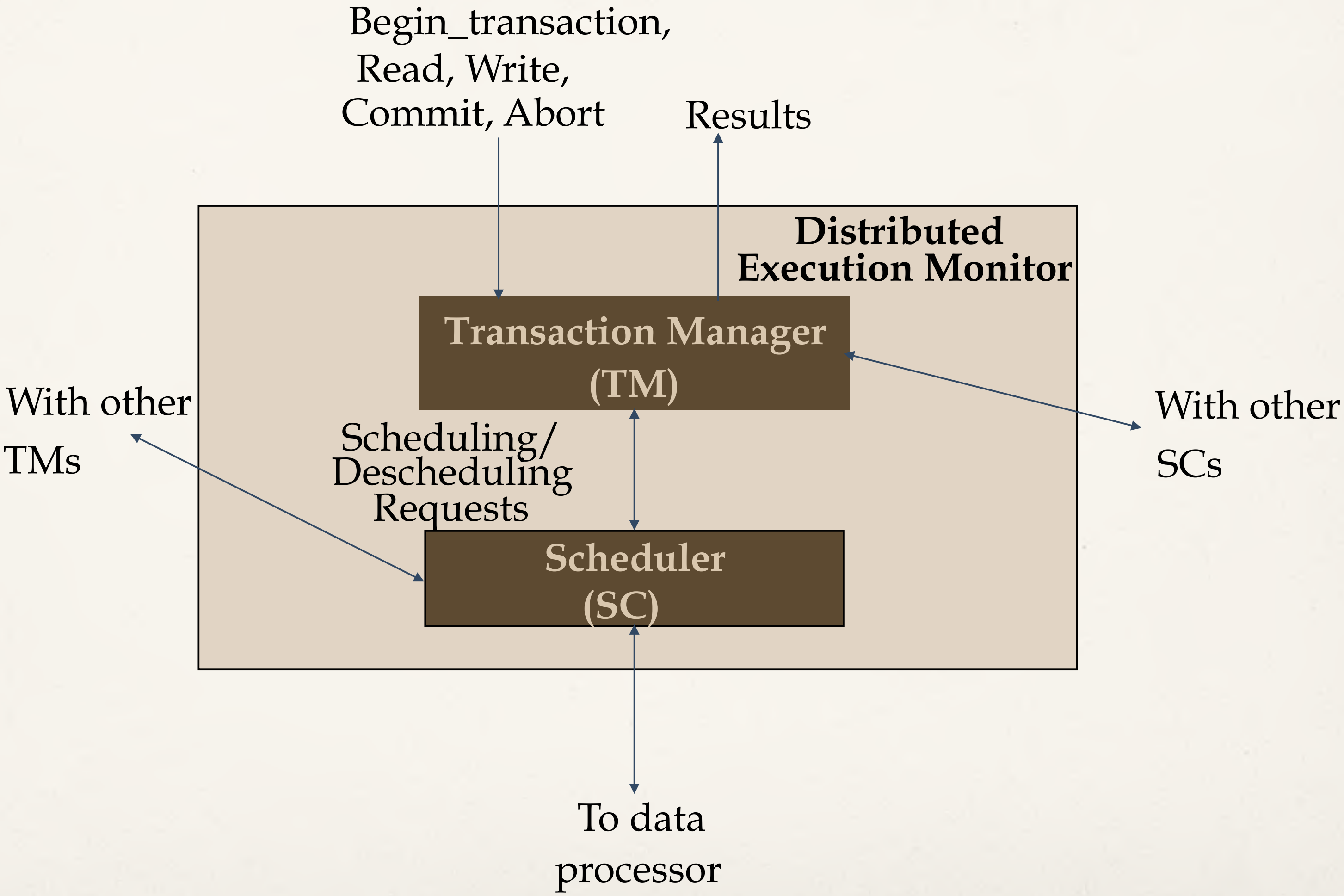
Transaction Processing Issues

- 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

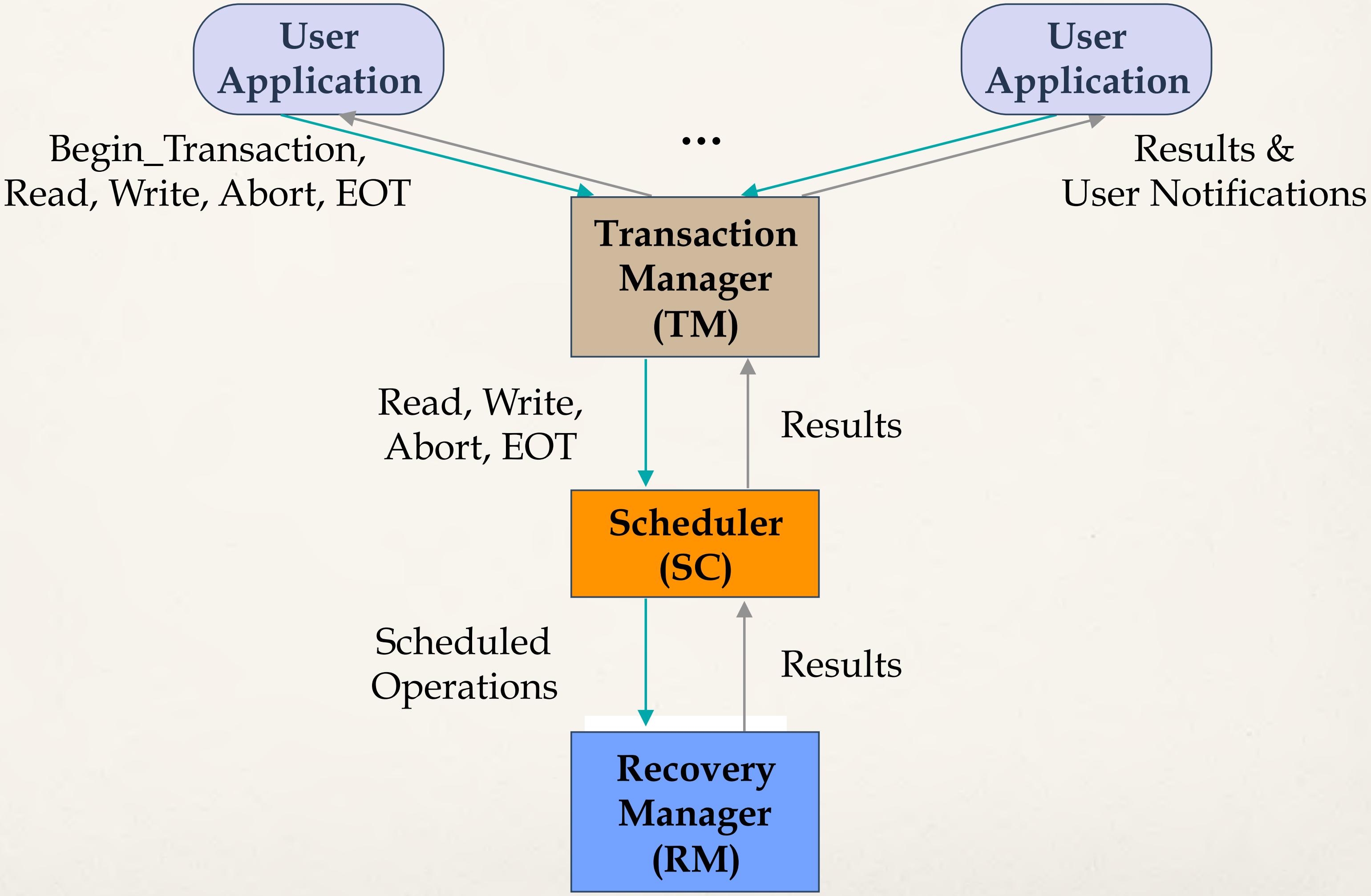
Transaction Processing Issues

- 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

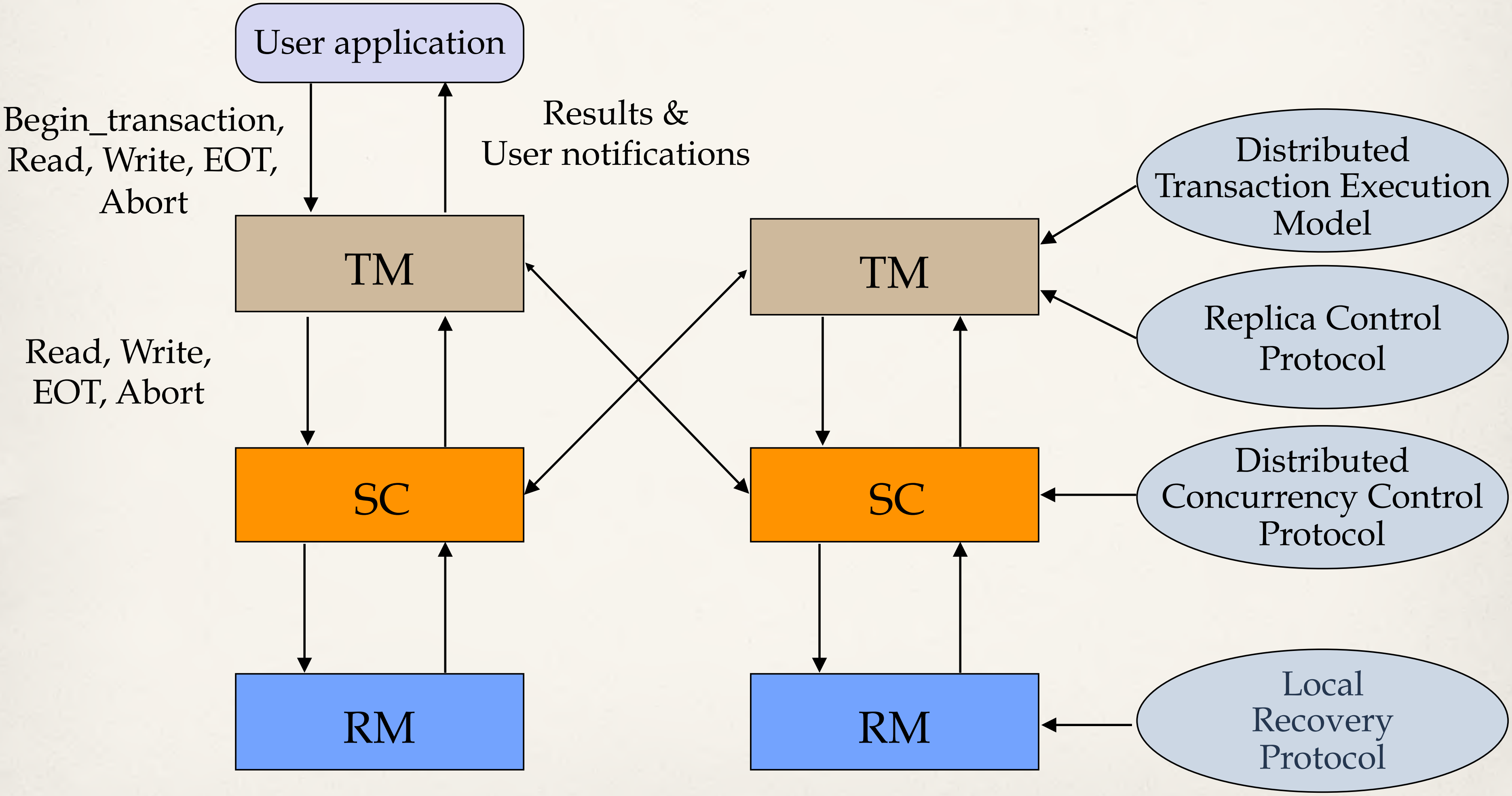


Centralized Transaction Execution



© M. T. Özsu & P. Valduriez

Distributed Transaction Execution


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

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.

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

T_2: Read(x)
Read(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

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 write on x

❹ Conflicting operations are resolved by timestamp order.

Basic T/O:

<u>for $R_i(x)$</u>

```
if $ts(T_i) < wts(x)$
then reject $R_i(x)$
else {accept $R_i(x)$,
$rts(x) \leftarrow ts(T_i)$}
```

<u>for $W_i(x)$</u>

```
if $ts(T_i) < rts(x)$ or $ts(T_i) < wts(x)$
then reject $W_i(x)$
else { accept $W_i(x)$,
$wts(x) \leftarrow ts(T_i)$ }
```

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

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

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

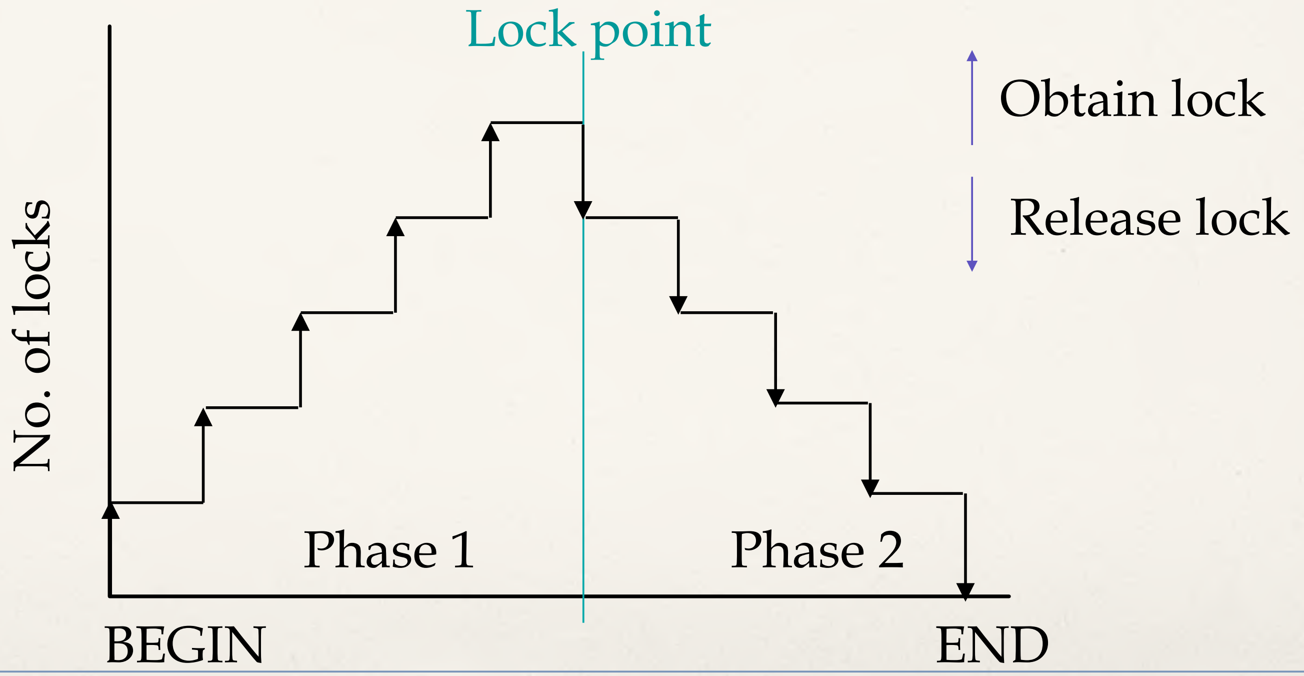
- Locking works nicely to allow concurrent processing of transactions.

Two-Phase Locking (2PL)

❶ A Transaction locks an object before using it.

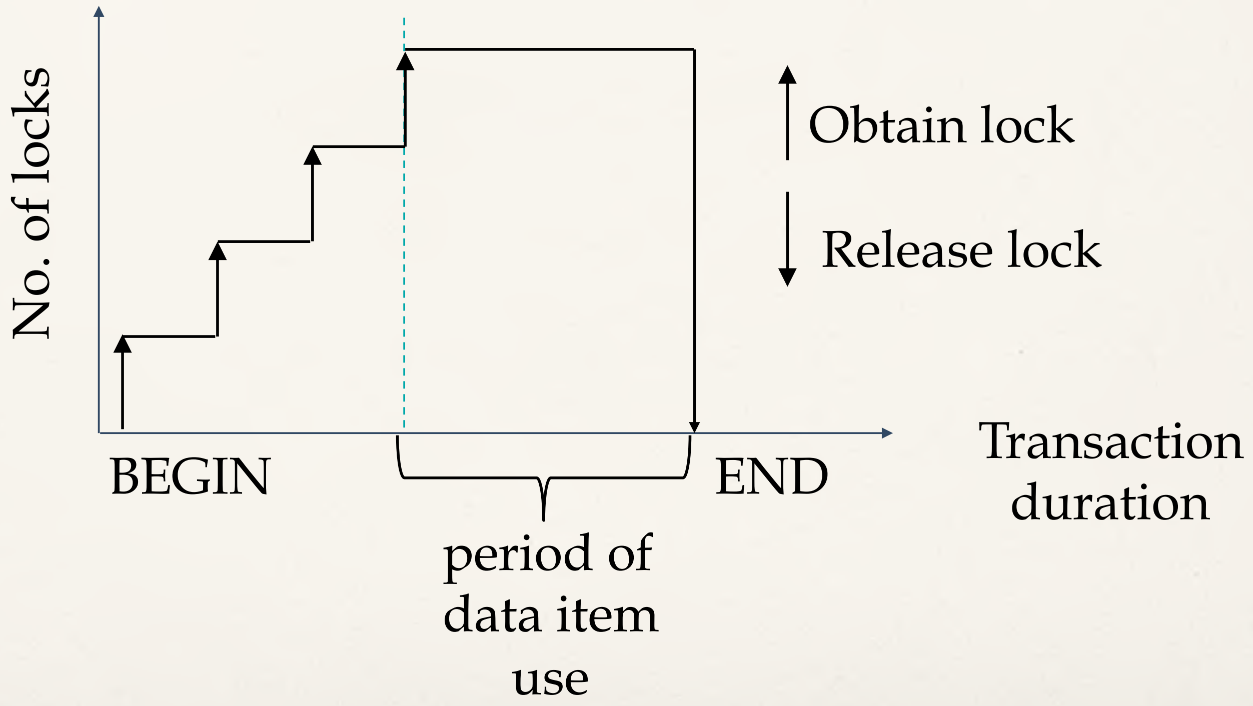
❷ When an object is locked by another transaction, the requesting transaction must wait.

❸ When a transaction releases a lock, it may not request another lock.



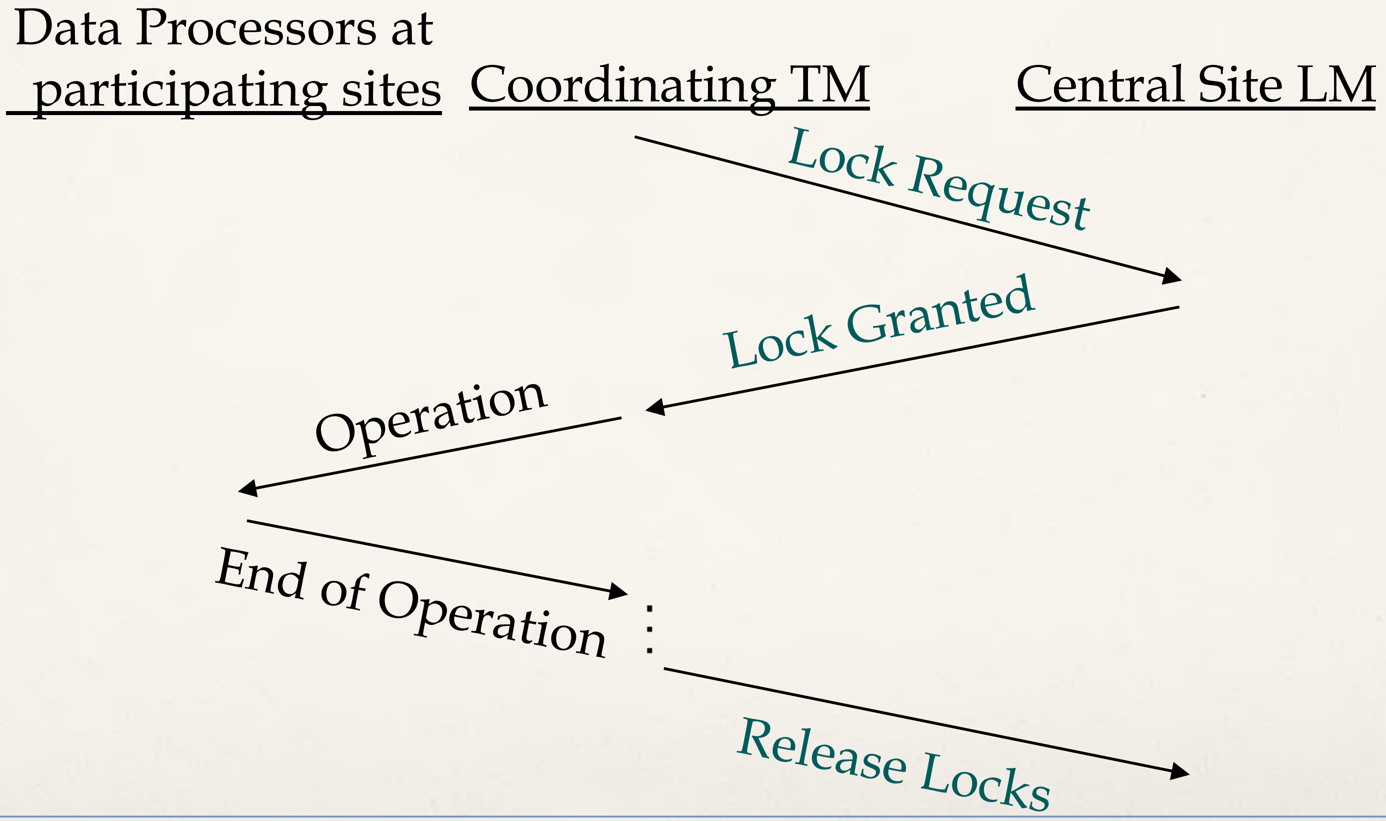
Strict 2PL

Hold locks until the end.



Centralized 2PL

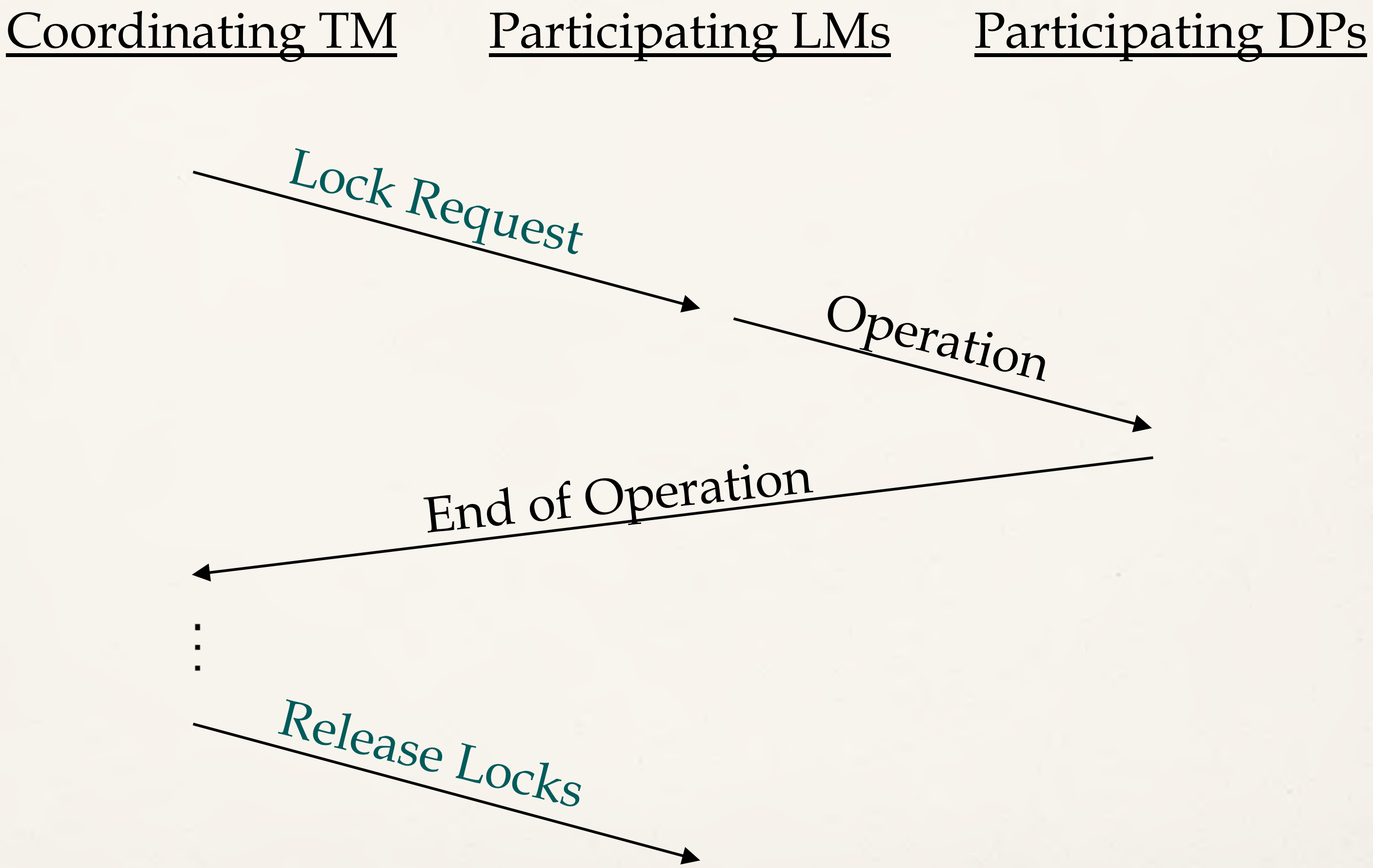
- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



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



Coordinating TM

Participating LMs

Participating DPs

Lock Request

Operation

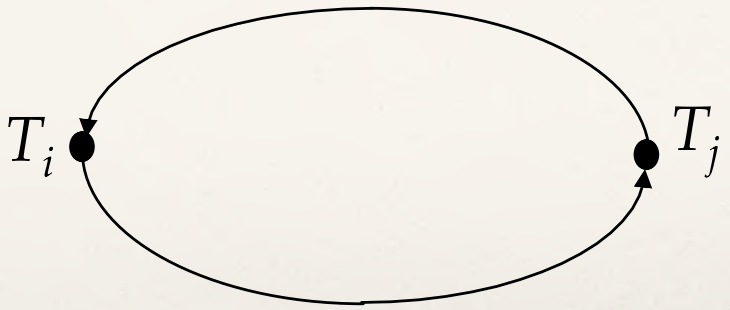
End of Operation

⋮

Release Locks

Deadlock

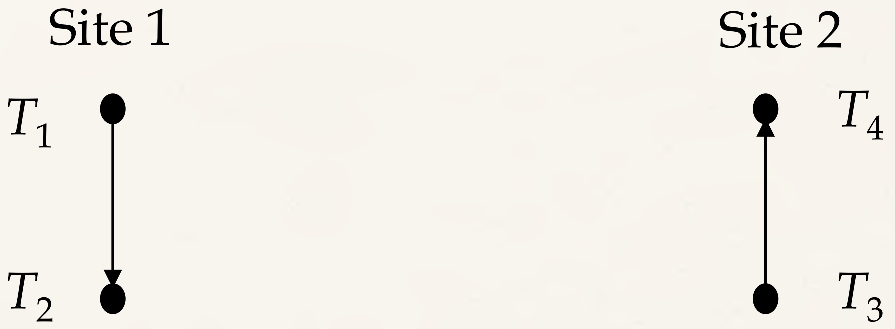
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- TO-based algorithms that involve waiting may cause deadlocks.
- Wait-for graph
  - ➡ If transaction T_i waits for another transaction T_j to release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



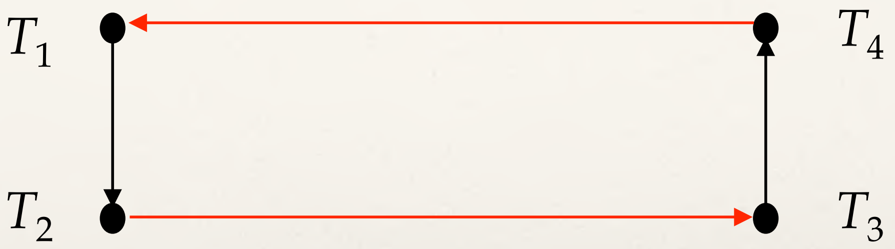
Local versus Global WFG

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 Detection

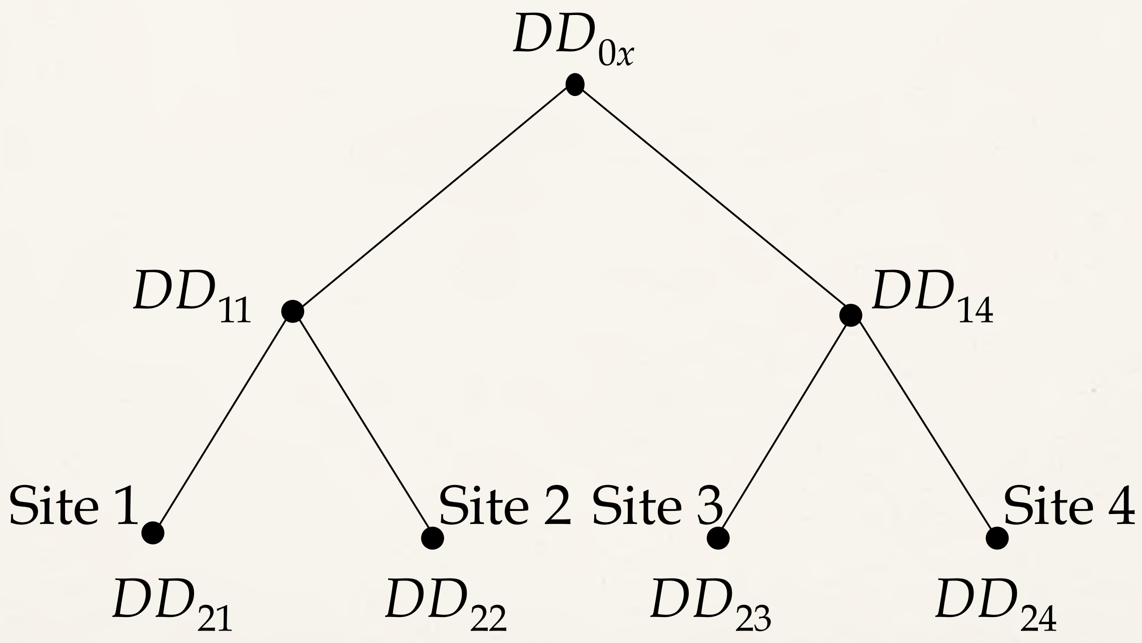
- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.
- Topologies for deadlock detection algorithms
  - ➡ Centralized
  - ➡ Distributed
  - ➡ Hierarchical

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

Reliability

Problem:

- How to maintain
  - atomicity
  - durability
- properties of transactions

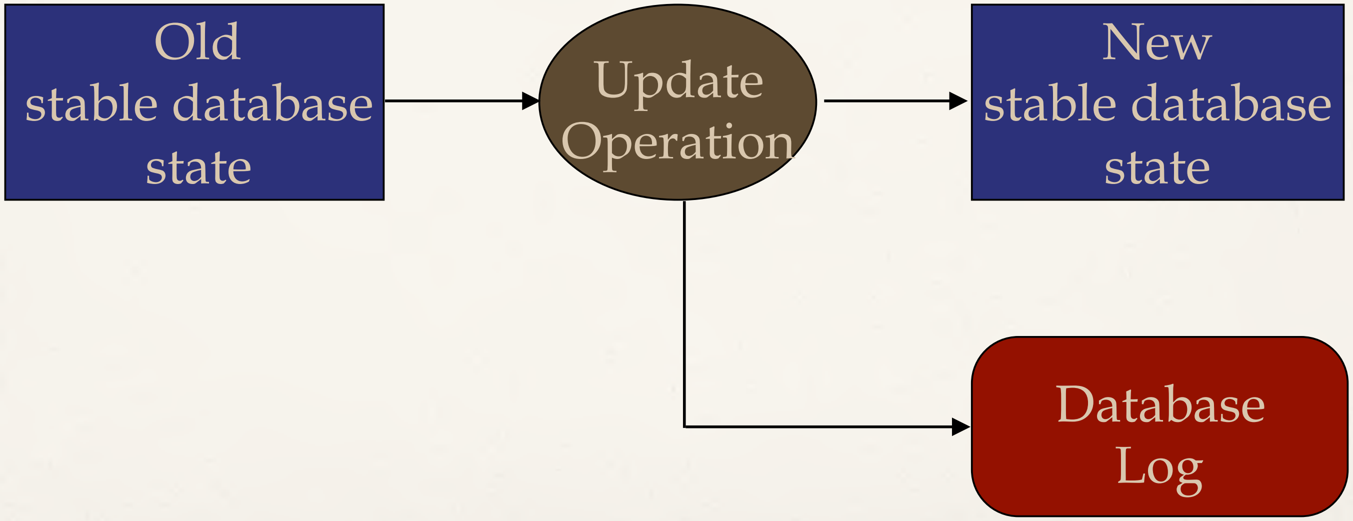
Update Strategies

- 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

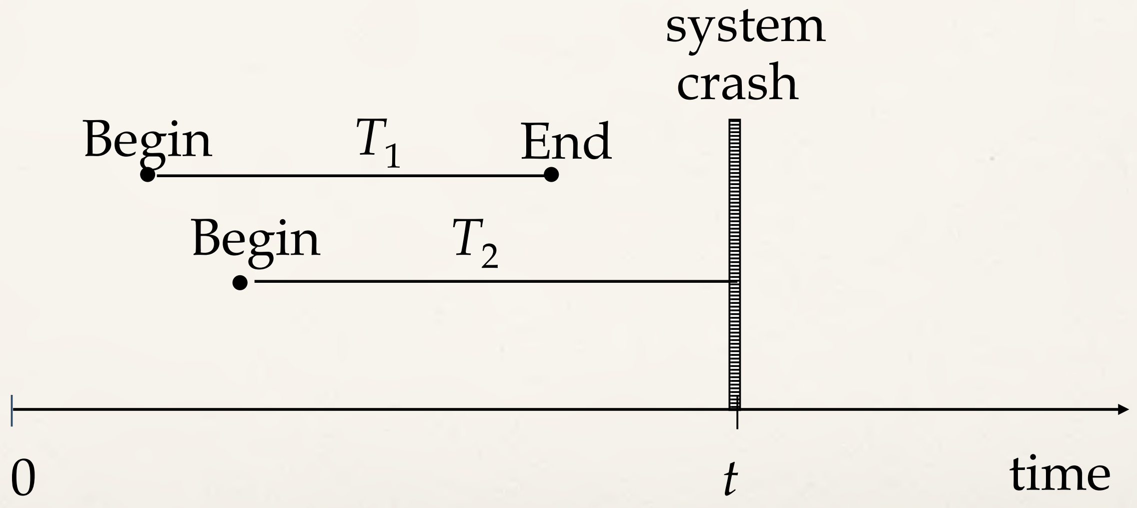
- ➡ transaction identifier
- ➡ type of operation (action)
- ➡ items accessed by the transaction to perform the action
- ➡ old value (state) of item (before image)
- ➡ new value (state) of item (after image)

…

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)



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

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

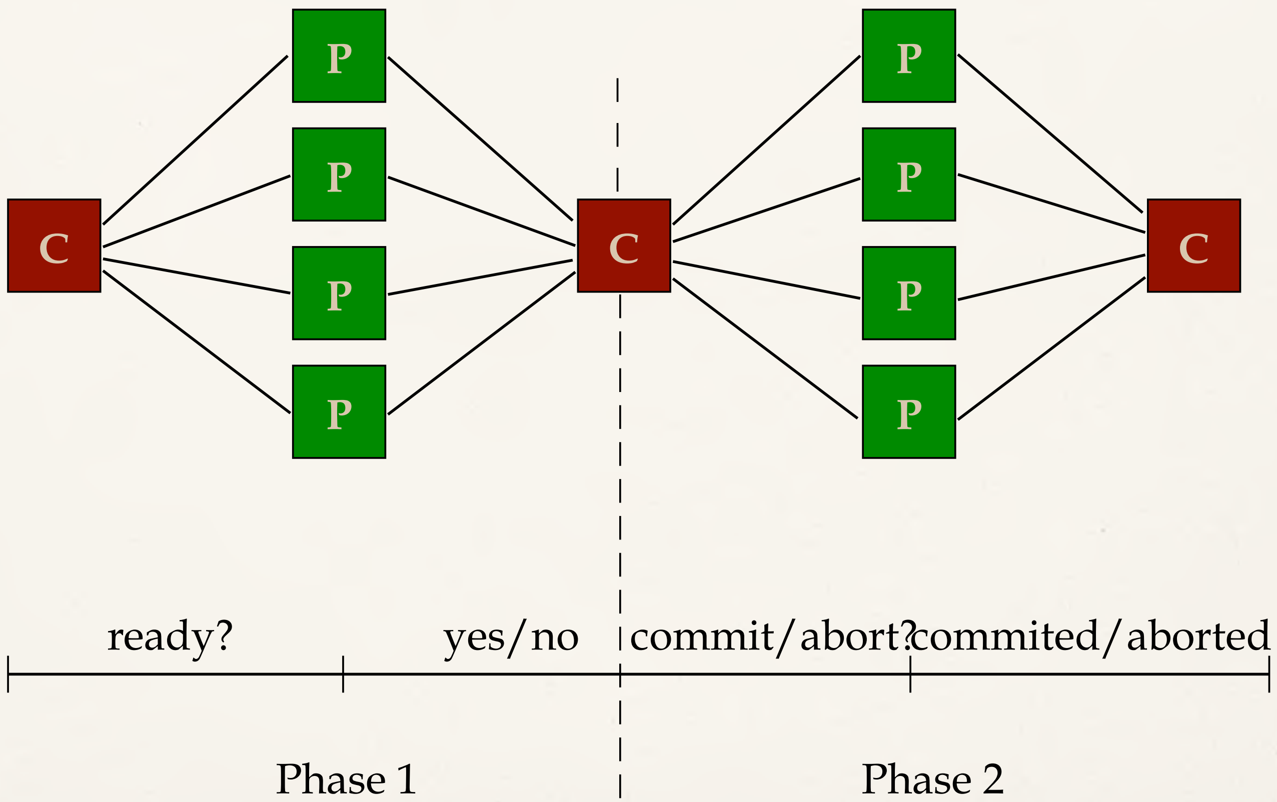
Phase 2 : Everybody writes the results into the database

- ➡ **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

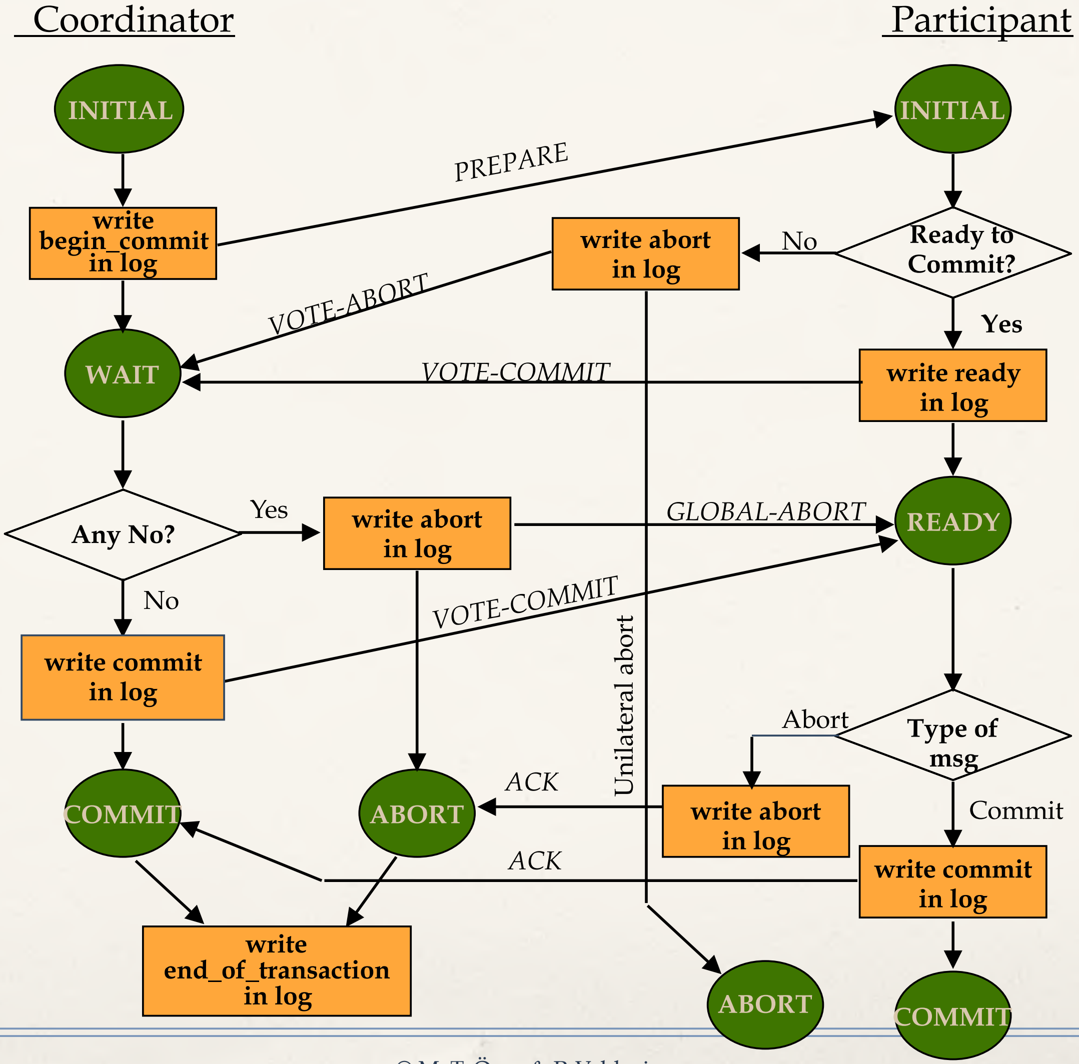
Global Commit Rule:

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.

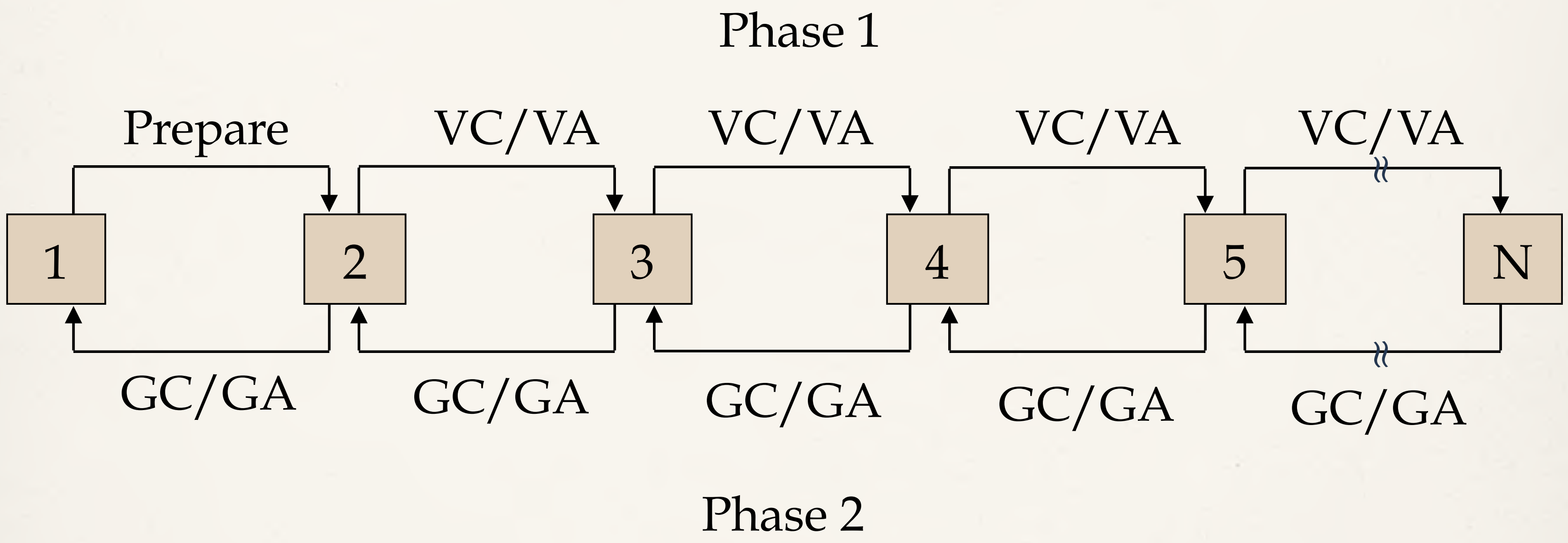
Centralized 2PC



2PC Protocol Actions

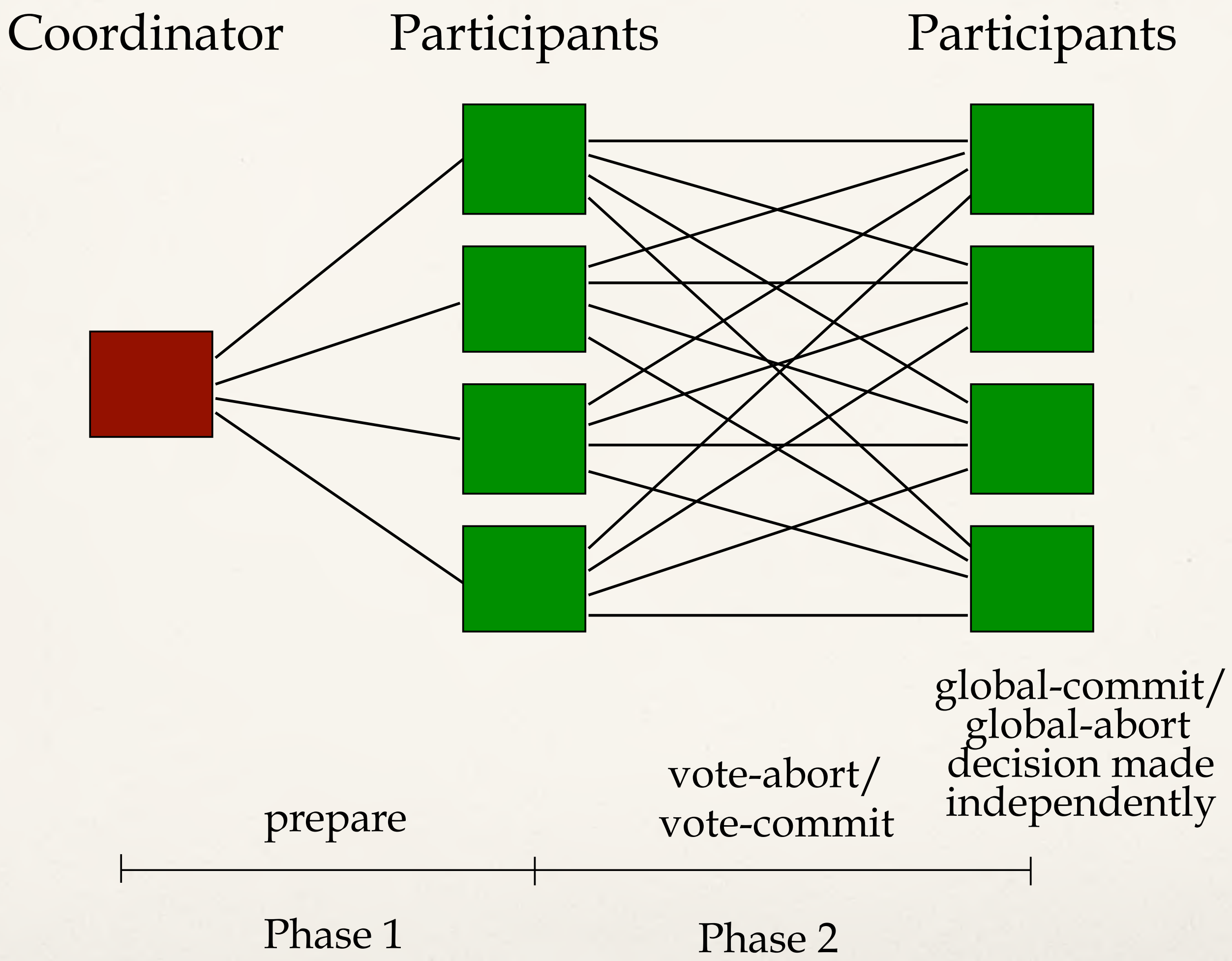


Linear 2PC



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

Distributed 2PC



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

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)