Distributed Transaction Management

Distributed Concurrency Control

© M. T. Özsu & P. Valduriez

Concurrency Control

Problem:

How to maintain

consistency

isolation

properties of transactions

Contents

- Serializability Theory

- Concurrency Control Algorithms (Taxonomy)
- Locking-Based Algorithms
- Deadlock Management
  - Deadlock Prevention, Avoidance, Detection and Resolution

© M. T. Özsu & P. Valduriez

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.

© M. T. Özsu & P. Valduriez

Concurrency control

Two main issues:

1. Correctness criterion for the concurrent execution of transactions
2. Algorithms to ensure that the criterion is verified

Spoiler: Serializable history and 2PL

History

- A history (schedule) is defined over a set of transactions $T=\{T_1, \ldots, T_n\}$ and specifies an interleaved order of execution of these transactions' operations.

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

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

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 conflicting operations (belonging to unaborted transactions) in the two histories are the same.

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

© M. T. Özsu & P. Valduriez

Serializability

- The primary function of the scheduler (SC) is to generate a serializable history for the execution of pending transactions.
- How? Concurrency control algorithm

© M. T. Özsu & P. Valduriez

Concurrency Control Algorithms

- Pessimistic
  - Two-Phase Locking-based (2PL)
    - Centralized (primary site) 2PL
    - **Distributed 2PL**
  - Timestamp Ordering (TO)
    - Basic TO
    - Multiversion TO
    - Conservative TO

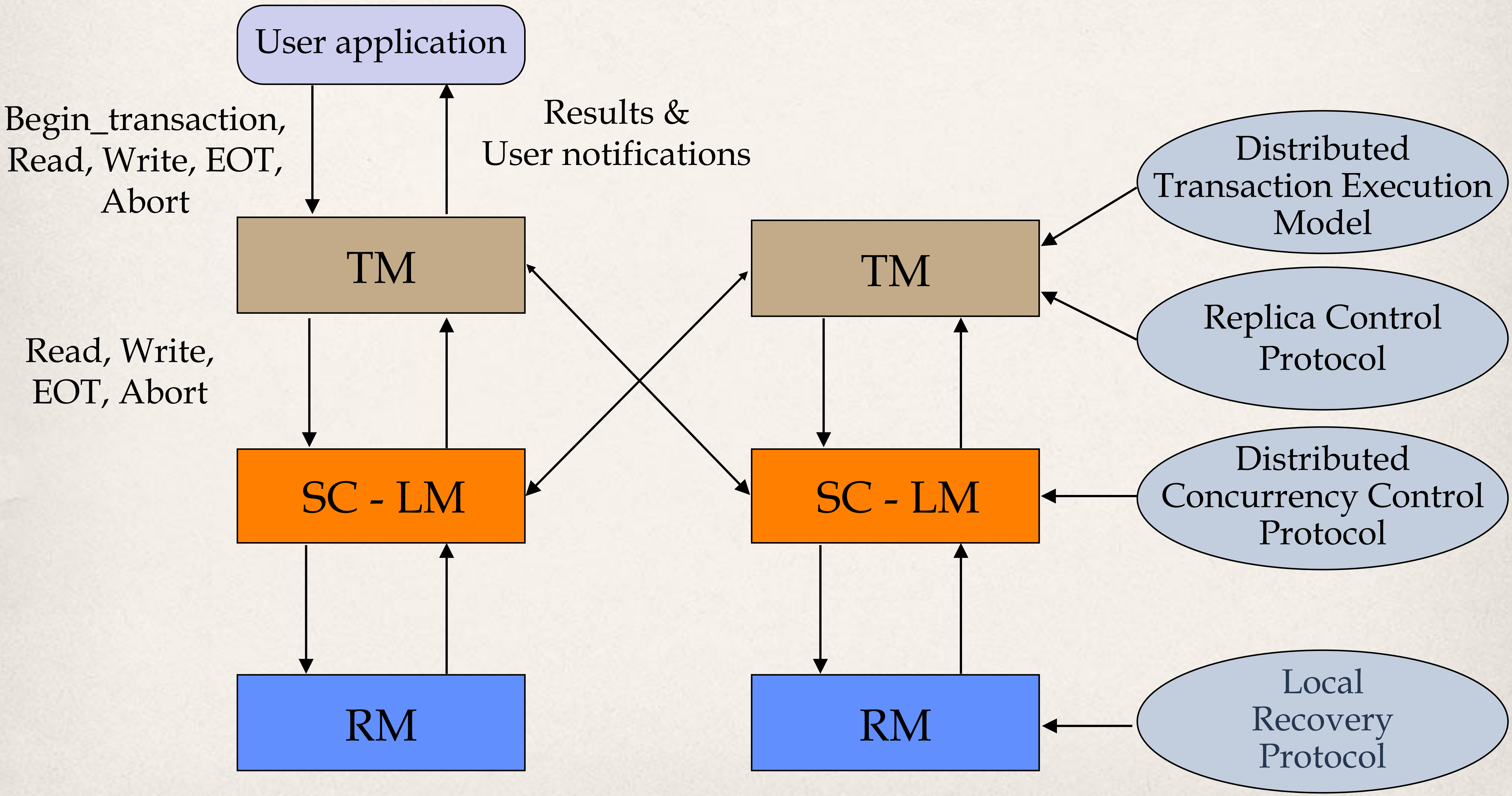
- Optimistic (not too many transactions will conflict with one another)
  - Locking-based
  - Timestamp ordering-based

Locking-Based Algorithms

- The main idea is to ensure that a data item that is shared by conflicting operations is accessed by one operation at a time.
- The synchronization of transactions is achieved by employing locks on some portion or granule of the database.
- The scheduler is also the lock manager (LM).

© M. T. Özsu & P. Valduriez

Distributed Transaction Execution



© M. T. Özsu & P. Valduriez

Locking-Based Algorithms

- 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), so only read locks are compatible.

	rl	*wl*
rl	yes	no
wl	no	no

Locking-Based Algorithms

1. A Transaction locks an object before using it.
2. When an object is locked by another transaction, the requesting transaction must wait.

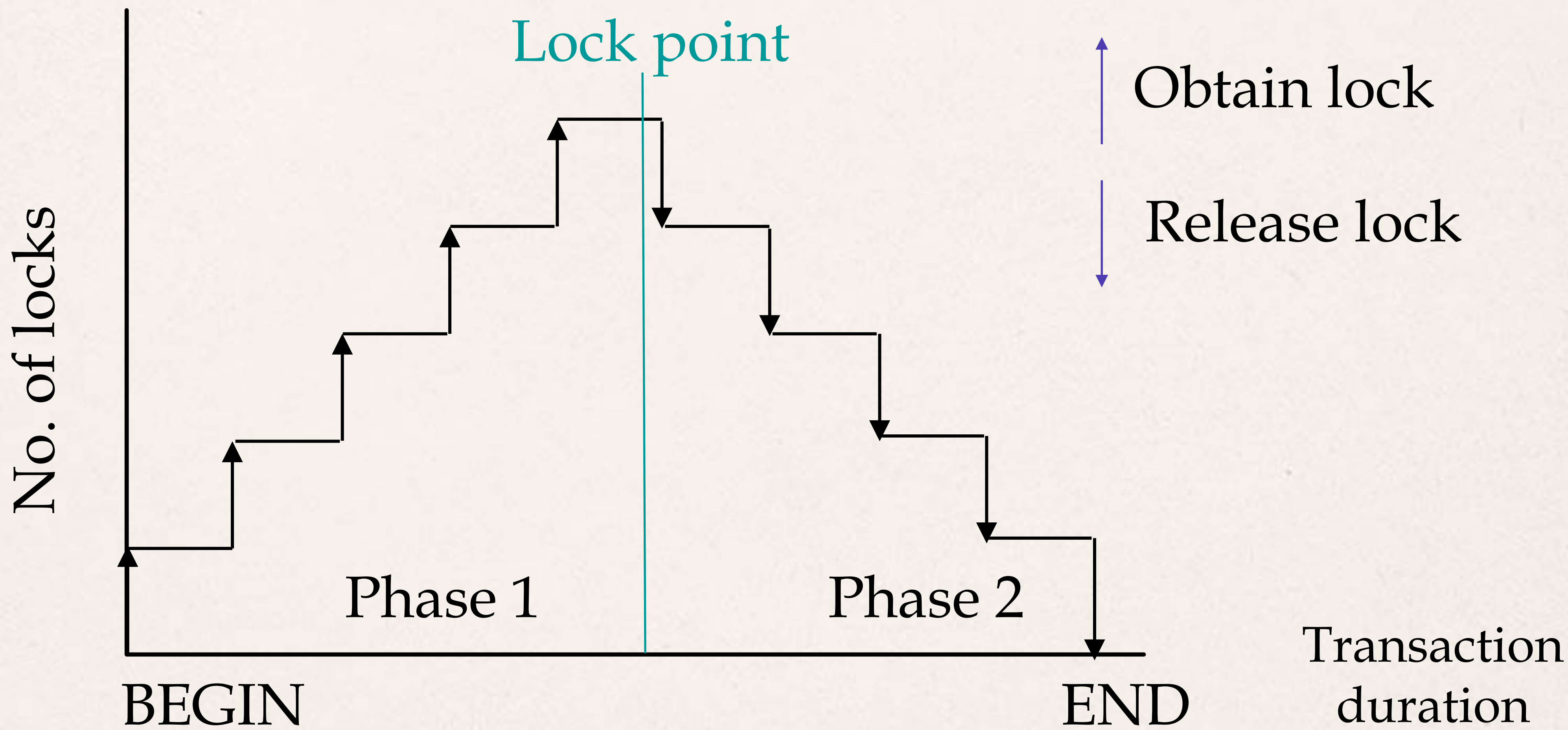
- Locking works nicely to allow concurrent processing of transactions.

  but we must add one more rule

© M. T. Özsu & P. Valduriez

Two-Phase Locking (2PL)

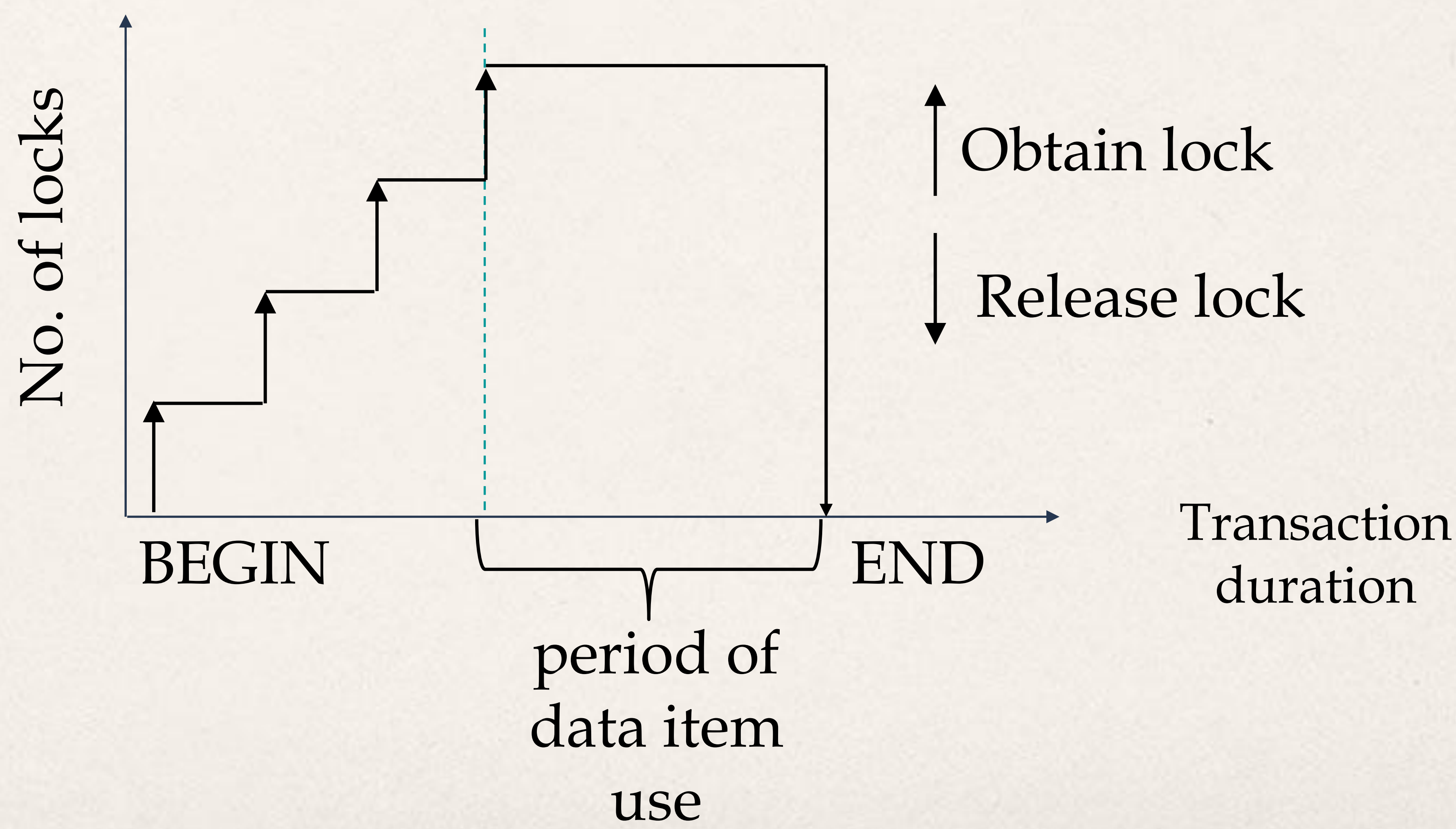
❸ **When a transaction releases a lock, it may not request another lock.**



© M. T. Özsu & P. Valduriez

Strict 2PL

Hold locks until the end.



© M. T. Özsu & P. Valduriez

Two-Phase Locking (2PL)

- It has been proven that any history generated by a concurrency control algorithm that obeys the 2PL rule is serializable.

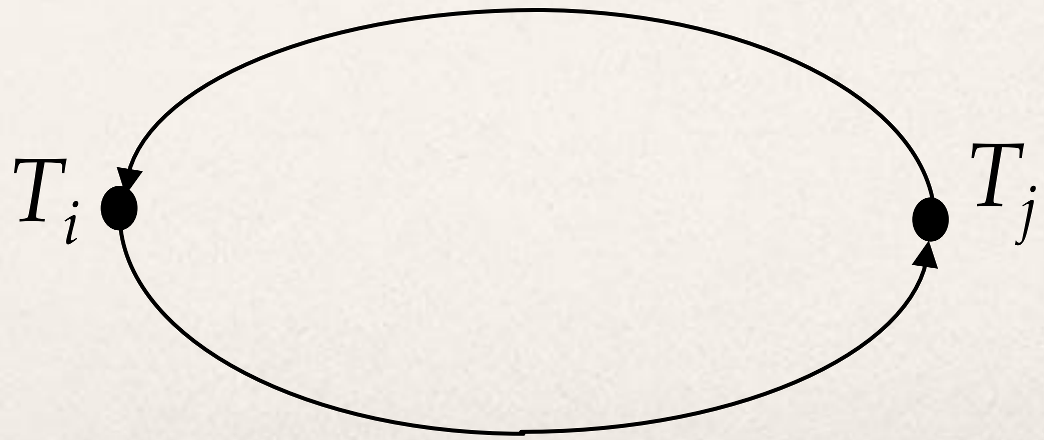
Safety

© M. T. Özsu & P. Valduriez

Deadlock

- Locking-based concurrency control algorithms may cause deadlocks.
- A transaction is deadlocked if it is blocked and will remain blocked until there is an intervention.

- 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**.



Liveness?

© M. T. Özsu & P. Valduriez

Distributed 2PL

- Symmetric 2PL schedulers are placed at each site.
- Each local scheduler manages locks for data at that site.

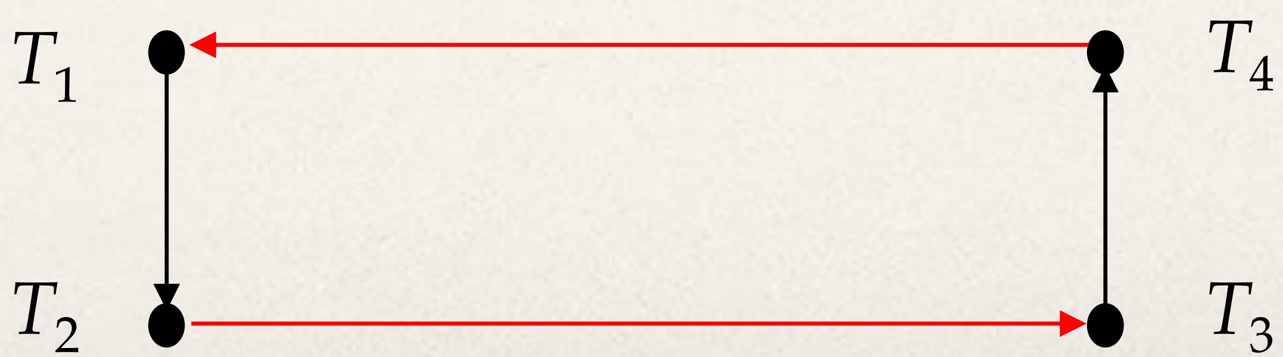
© M. T. Özsu & P. Valduriez

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 ensure that deadlock will not occur. Requires run time support.
- Detection and resolution
  - Allowing deadlocks to form and then finding and breaking them. Requires run time support.

© M. T. Özsu & P. Valduriez

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.
  - → Suitable for systems that have no provisions for undoing processes.
  - → Unsuitability of the scheme in a database environment.
- 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 (locked).
- Use timestamps to prioritize transactions and avoid deadlocks by aborting transactions with higher (or lower) priorities.
- More attractive than prevention in a database environment.

© M. T. Özsu & P. Valduriez

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 if and only if T_i is older than T_j.
- If T_i is younger than 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

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 if and only if T_i is younger than T_j.
- If T_i is older than 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 T_j is younger)

Deadlock Detection

- Transactions are allowed to wait freely.
- Wait-for graphs and cycles.

- Methods for detecting distributed deadlocks
  - Centralized
  - Hierarchical
  - **Distributed**

© M. T. Özsu & P. Valduriez

Distributed Transaction Management

Distributed Concurrency Control

© M. T. Özsu & P. Valduriez

Formalization of History

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

❶ $\sum_T = \bigcup_i \sum_i$ for $i = 1, 2, \ldots, n$

❷ $\prec_H \supseteq \bigcup_i \prec_{T_i}$ for $i = 1, 2, \ldots, n$

❸ For any two conflicting operations $O_{ij}, O_{kl} \in \sum_T$, either $O_{ij} \prec_H O_{kl}$ or $O_{kl} \prec_H O_{ij}$

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 history is given as the DAG

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

© M. T. Özsu & P. Valduriez

Schedule

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

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

Serializability in Distributed DBMS

- Somewhat more involved. Two histories have to be considered:
  - local histories
  - global history
- For a 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 (sites) where they appear together.

© M. T. Özsu & P. Valduriez

Global Non-serializability – Example

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 (indeed, they are serial), but the two transactions are not globally serializable.

LH_1=$\{R_1(x),W_1(x), R_2(x)\}$ $\quad$ $T_1 \rightarrow T_2$

LH_2=$\{R_2(y), R_1(y),W_1(y)\}$ $\quad$ $T_2 \rightarrow T_1$

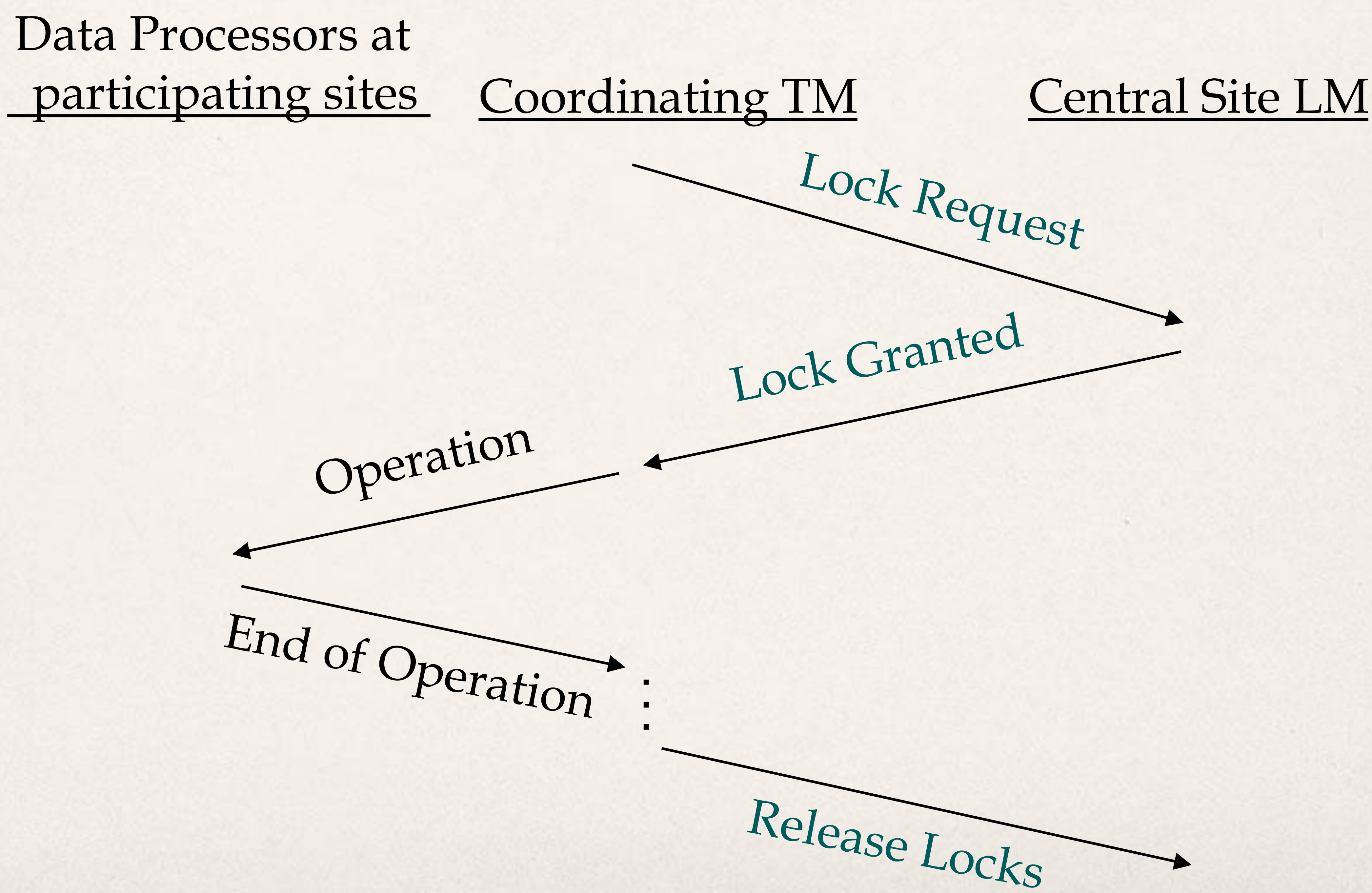
© M. T. Özsu & P. Valduriez

Centralized 2PL

- There is only one 2PL scheduler in the distributed system.
- Locks are managed by the central scheduler (primary site).

© M. T. Özsu & P. Valduriez

Centralized 2PL-TM Execution

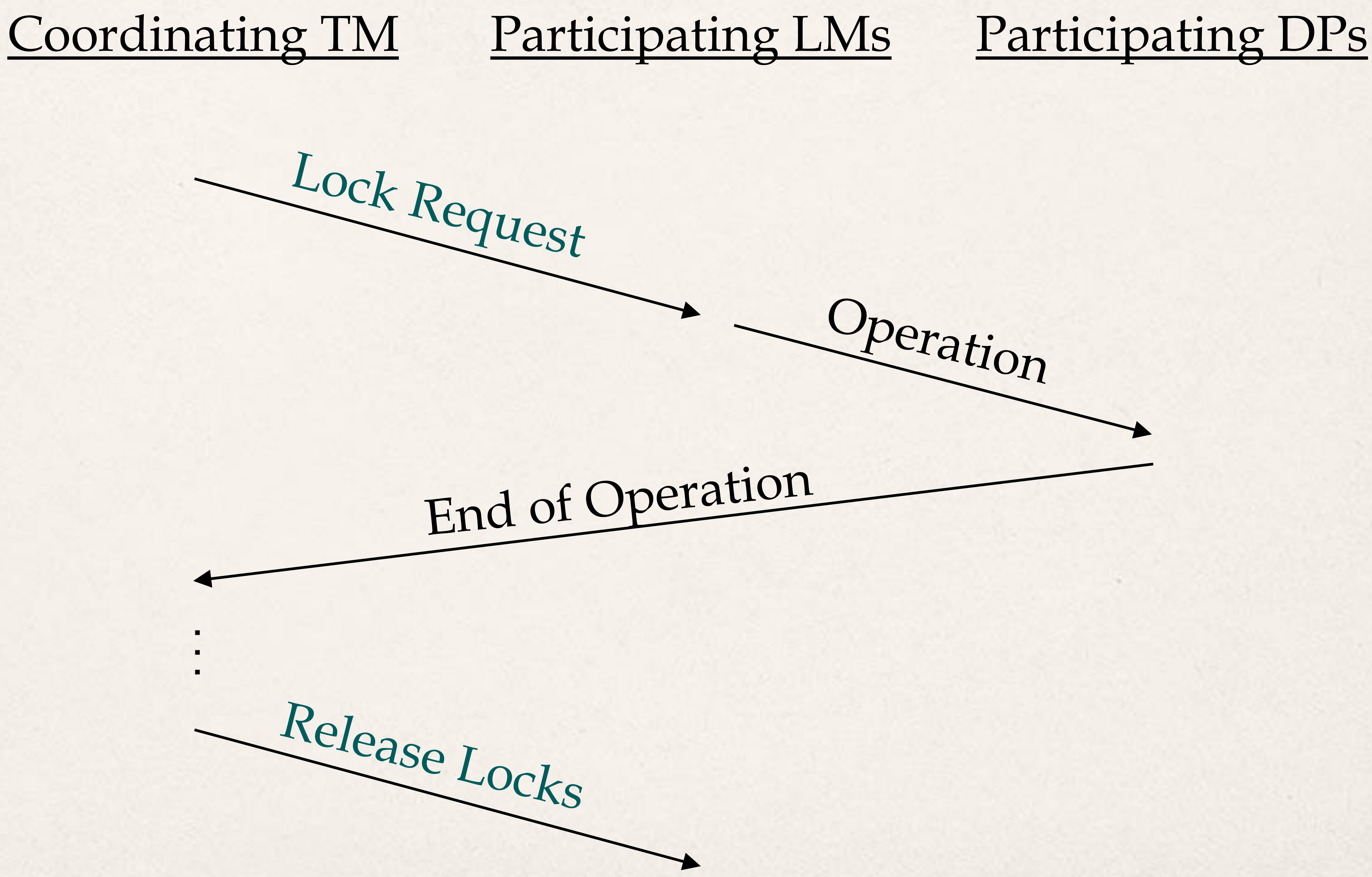


Distributed 2PL

- Symmetric 2PL schedulers are placed at each site.
- Each local scheduler manages locks 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.

© M. T. Özsu & P. Valduriez

Distributed 2PL Execution



© M. T. Özsu & P. Valduriez

Timestamp-Based Algorithms

- Timestamp-based concurrency control algorithms select, a priori, a serialization order and execute transactions accordingly.

- To establish this ordering, the transaction manager assigns each transaction T_i a globally unique timestamp, ts(T_i), at its initiation.

- **TO Rule**. Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k, O_{ij} is executed before O_{kl} if and only if ts(T_i) < ts(T_k). In this case T_i is said to be the older transaction and T_k is said to be the younger one.

© M. T. Özsu & P. Valduriez

Basic Timestamp Ordering

1. Transaction (T_i) is assigned a timestamp $ts(T_i)$.
2. Transaction manager attaches the timestamp to all operations issued by the transaction.
3. 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
4. Conflicting operations are resolved by timestamp order.

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

if $ts(T_i) < wts(x)$
then reject $R_i(x)$
else accept $R_i(x)$
$\quad rts(x) \leftarrow ts(T_i)$

<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 $W_i(x)$
$\quad wts(x) \leftarrow ts(T_i)$

© M. T. Özsu & P. Valduriez

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

© M. T. Özsu & P. Valduriez

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, so that $ts(x_w) = ts(T_i)$, if the scheduler has not yet processed any $R_j(x_r)$ such that

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

© M. T. Özsu & P. Valduriez

Optimistic Concurrency Control Algorithms

Pessimistic execution

Validate | Read | Compute | Write

Optimistic execution

Read | Compute | Validate | Write

© M. T. Özsu & P. Valduriez

“Relaxed” Concurrency Control

- Non-serializable histories
  - Semantics of transactions can be used
    - Look at semantic compatibility of operations rather than simply looking at reads and writes

© M. T. Özsu & P. Valduriez