

# *Replicated Data Management*

Luc Onana Alima  
KTH/IMIT/LECS  
Spring 2003

# Outline

---

- ✍ Introduction
- ✍ The concept of transaction
- ✍ Transaction processing systems
- ✍ Database Techniques
- ✍ Atomic Multicast
- ✍ Update Propagation

# Introduction

---

## Database

- Set of named data items
- Transactional accesses

## Replicated database

- Distributed database
- Multiple copies of the same item at different nodes

## Why replicating data

- Increase availability
  - Increase the chance to access a needed data item, even though some nodes are failed
- Improve performance
  - Increase the chance to access close by copies

# The concept of transaction: Usefulness (1/3)

---

## Assume

- We want to transfer 1000SEK from account A to account B
- Balance of A is greater than 1000SEK

## Operations involved

- Op1: Subtract 1000SEK from account A
- Op2: Add 1000SEK to account B

# The concept of transaction: Usefulness (2/3)

---

- ✍ The situation where for example, only Op1 is performed is not desirable
- ✍ Transactions serve to avoid such situations
- ✍ The concept of transaction was originally introduced to protect the integrity of the data in centralized multi-user databases
- ✍ Next, the concept proved useful for fault-tolerance

# The concept of transaction: Usefulness (3/3)

---

✍ To achieve fault-tolerance, a transaction processing system should be recoverable, or able to restore a consistent state after a failure

## ✍ Transaction

- Sequence of operations on data items
- This sequence of operations has the ACID property
- So, what does ACID mean?

# Transaction: Properties

---

## **Atomicity**

- Either all operations succeed or none of them are, in spite failures

## **Consistency**

- From consistent state to consistent state

## **Isolation**

- Intermediate states produced by a transaction are not visible to others before the transaction is successfully committed

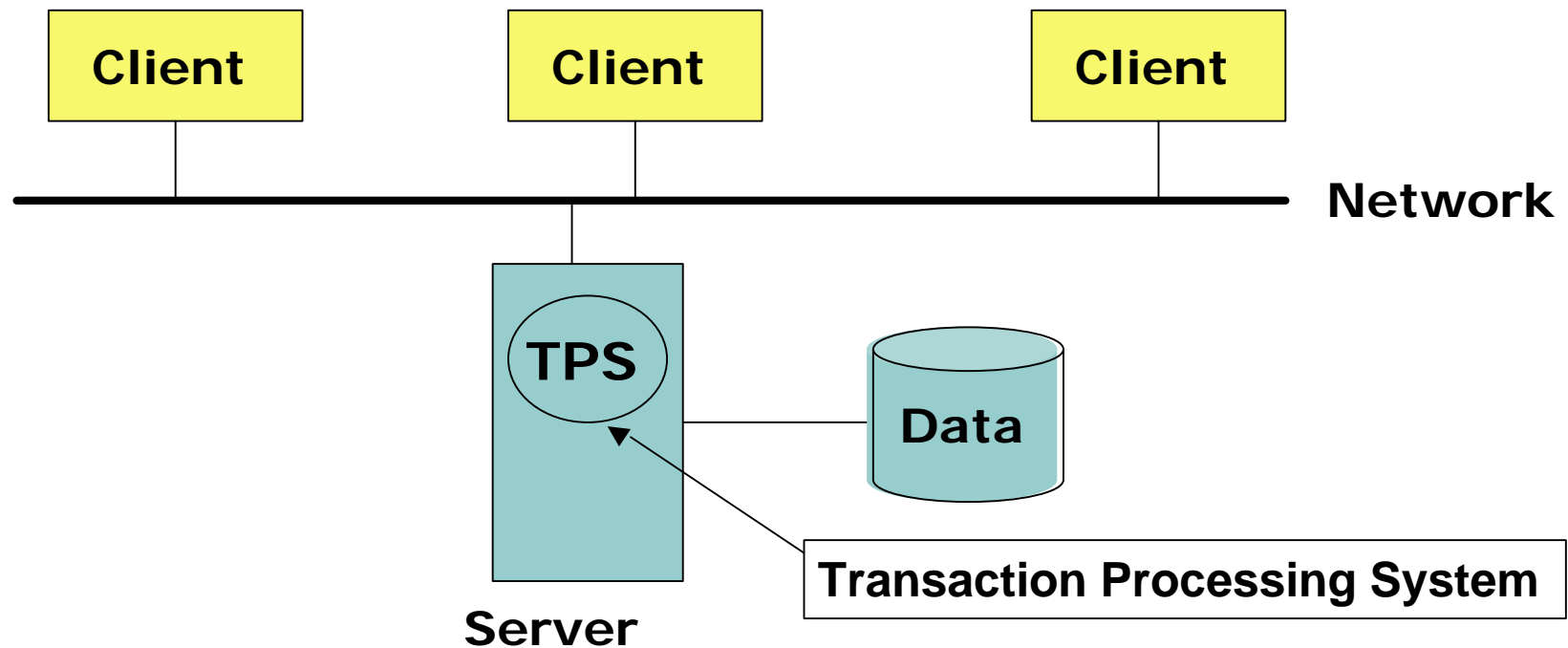
## **Durability**

- Effects of successful transactions are made permanent

# Centralized context: Client-Server

---

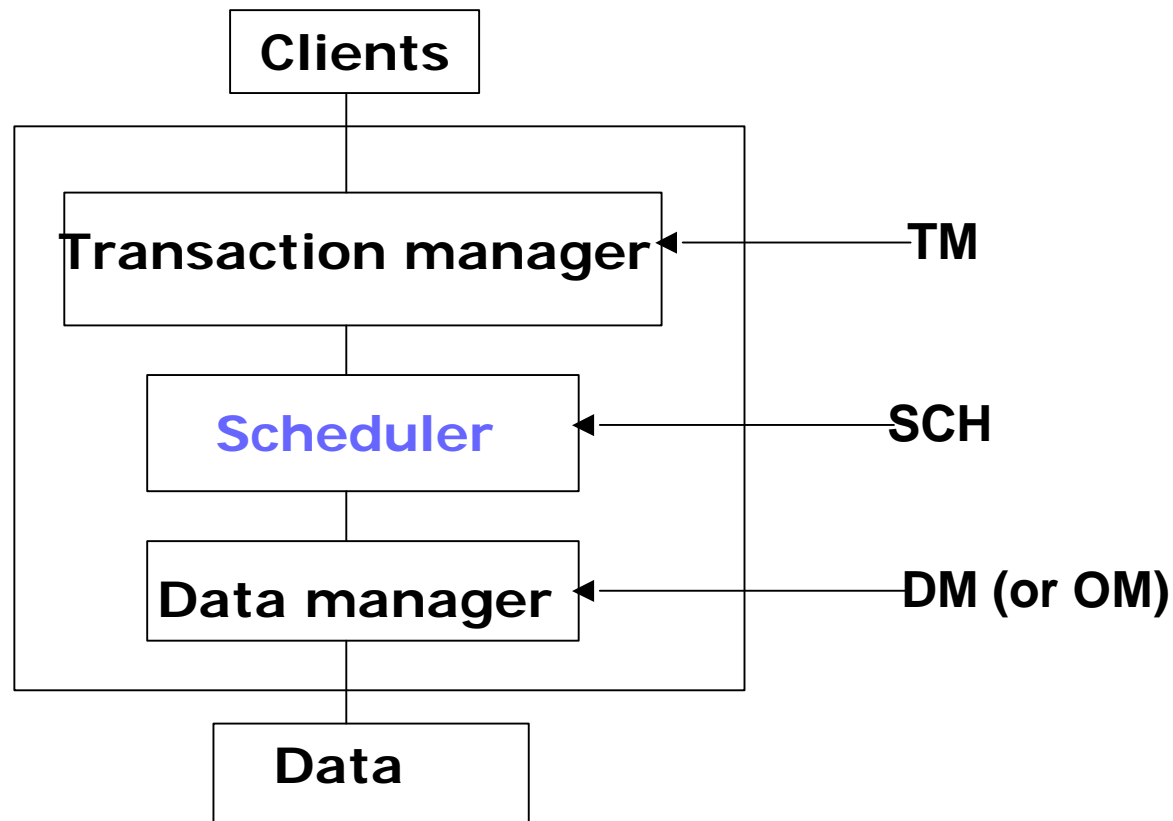
- ✍ Several clients access the database concurrently



# Transaction Processing System: Centralized context

---

✍ A simplified architecture



# Transaction Processing System: Distributed context

---

## ✍ In distributed context

- Several nodes

## ✍ The distributed Transaction Processing System (DTPS)

- One TPS at each node

Need for coordination/cooperation between major components of a TPS

# Distributed Transaction Processing System

---

## TM

- Does some preprocessing of transactions
- Can serve to locate copies of items
- Involves in **atomic commitment protocol**
- Interface between the TPS and the clients

## SCH

- Ensure maximum concurrency without interference
- Uses **concurrency control algorithms**

## DM

- Interface to the actual data
- Ensure recovery for durability
- Uses **replicated data management protocols** for consistency of replicas

# Replicated Data Management System (RDMS)

---

- ✍ Hides replication to users
  - From users standpoint, there is no replication
- ✍ Translate read/write on logical items to read/write on copies of items
- ✍ Uses concurrency control algorithms to synchronize access to copies
  - E.g. Two phase locking, Timestamp-based

# Database techniques: Outline

---

- ✍ Introduction to logging and recovery
- ✍ Two-phase commit
- ✍ Three-phase commit
- ✍ **Serializability**
  - Static quorum
  - Dynamic quorum changes

# Introduction to recovery: Database logging and recovery

---

- ✍ During its execution, a transaction T will:
  - Read data items in RT, the read set of T
  - Write into data items WT, the write set of T
  - These data items are located at processors in PT, the processor set of T
- ✍ The recovery should prevent partial update of WT from being visible to others if transaction T fails before it completes
- ✍ **The question is how recovery works ?**
  - We give the idea on centralized system to start

# Recovery on a centralized system: Redo-only approach

---

- ✍ When T submits Update(w) to the Database System (DBS),
  - The update is not performed immediately by the DBS
  - Instead, the DBS records the proposed update in the T's **intention list**
- ✍ When T completes, it makes a decision to **commit**

# Recovery on a centralized system: Redo-only approach

---

- ✍ Before T is allowed to commit, the following steps are performed
  - Step 1: The contents of the intention list is written to the database log file
    - This is the **precommit** of T
  - Step 2: A commit record is written to the log file
    - This is the **commit point**
  - Step 3: The contents of the intention list is applied to the database

# Recovery on a centralized system: Redo-only approach

---

## **Failure handling**

- If a failure occurs before step 2
  - The DBS can restart in a state in which T never executed
- If a failure occurs after step 2
  - The updates can be applied to the database (the updates are taken from the log) if they have not yet been applied

# Recovery on a centralized system: Redo-only approach

---

## Atomicity

- Is ensured, no matter when a failure occurs, either all updates that a transaction makes are recorded in the database, or none of the updates are recorded

## Recovery on a centralized system: Undo-only approach

---

- ✍ When a transaction  $T$  wants to update  $w$ ,
  - The old value of  $w$  is recorded
  - The update is performed immediately
- If  $T$  commits, the DBS throws away the old value of  $w$
- If  $T$  aborts, the old value of  $w$  is restored

# Two-phase commit: Objective and terminology

---

✍ Used to get all the processors involved in the distributed transaction to agree to commit or abort

## ✍ Terminology


- Let  $T$  be a distributed transaction
- Coordinator: processor that initiated  $T$
- Participants: all the other processors involved in the processing of  $T$

# Two-phase commit: Assumptions and requirements (1/2)

---

## Failure assumption

- Fail-stop processor
- Communication can fail (but we will focus on proc. Failure only)
- Failures are eventually repaired
- Timeout serve to detect failures

 We assume that each processor follows the logging protocol as in the Redo-only recovery

## Two-phase commit: Assumptions and requirements(2/2)

---

- ✍ Required is a protocol that satisfies the following
  - All processors reach the same decision (Commit or Abort)
  - A processor cannot change its decision once made
  - If no failure occurs and all processors can commit, then the decision is “Commit”
  - If all processors can commit, and all (tolerable) failures are repaired and no new failures occur, then all processors eventually commit

# Two-phase commit: The algorithm (1/3)

---

## Coordinator

- Precommits the transaction
- Sends `VoteRequest` to all participants and starts to wait for votes (Yes or No)
- If all participants voted YES and the coordinator vote is Yes, then the coordinator commits (i.e. write Yes and commit record to log file) and sends a `Commit` message to all participants
- Otherwise, the coordinator aborts the transaction and sends an `Abort` message to all participants

# Two-phase commit: The algorithm (2/3)

---

## Participant p

- Initially waits for a VoteRequest from coordinator
- Upon receipt of VoteRequest from the coordinator
  - If p can commit then
    - p precommits the transaction
    - p writes a “Yes” record to the log file
    - p sends “Yes” vote to coordinator
  - Otherwise
    - p aborts the transaction
    - p sends “No” to coordinator
  - p waits for the decision

# Two-phase commit: The algorithm (3/3)

---

## Participant p

- Upon receipt of Commit from the coordinator
  - p commits the transaction
- Upon receipt of Abort from coordinator
  - p aborts the transaction

# Two-phase Commit: Observations

---

✍ Several places where a participant waits for a message to arrive

- So, should it wait forever? No, after timeout it should do something!
  - We need to add timeout actions

✍ Yes, failures can occur and eventually repaired

- So, what to do when recovering from failure?
  - We need recovery protocols

✍ To sum up, we need to handle failures!

# Two-phase commit: Failure handling

---

✍ Adding timeout actions

✍ Recovery protocol

- What a processor does upon recovery from a failure
- The actions taken by a processor when it recovers from a failure depends on the **state of the transaction** at the time of failure of this processor

✍ The recovery protocol serves to detect stable property: “all voted Yes or not”

# Two-phase commit: Timeout actions

---

- ✍ Coordinator timeout waiting for vote
  - The transaction is aborted
- ✍ Participant timeout waiting for VoteRequest
  - The transaction is aborted
- ✍ Participant voted Yes and timeout waiting for the decision
  - Participant must contact the others, **termination protocol**
- ✍ Participant voted No and timeout waiting for decision
  - The transaction is aborted

# Two-phase commit: State of transaction (1/2)

---

- ✍ What a processor does upon recovering from a failure depends on the state of the transaction
  
- ✍ **State of a transaction is one of**
  - **Executing**
    - From the time of its initiation to the commit point
  - **Committed**
    - After the commit record is written to the log file
  - **Aborted**
    - After the abort record is written to the log file
  - **Uncertain**
    - When the two-phase commit protocol is being executed

## Two-phase commit: State of a transaction (2/2)

---

- ✍ An executing transaction is characterized by
  - No Abort, Yes or Commit in the log file
- ✍ An uncertain transaction is characterized by
  - **Yes** record in the log file
  - No Commit record in the log file
  - No Abort record in the log file

## Two-phase commit: Recovery protocol (1/2)

---

- ✍ When a processor recovers from a failure
  - It aborts all non-precommitted transactions
  - All processors will abort
  
- ✍ When the **coordinator** recovers from a failure that occurred **after** precommit but **before** Commit or Abort
  - It can abort the transaction

## Two-phase commit: Recovery protocol (2/2)

---

- ✍ When a **participant** recovers from a failure that occurred **after** precommit but **before** Commit or Abort
  - It cannot unilaterally decide, it must contact the others processors to decide, i.e. run a termination protocol

## Two-phase commit is blocking

---

- ✍ Even with the termination protocol, blocking situations can occur
  - Indeed, if all participants have voted Yes and coordinator fails before sending the decision, all participants are uncertain
  - They have to wait until the coordinator comes up again!

# Three-phase commit

---

## Failure assumption

- Fail-stop
- Timeout to detect failures

## Motivation

- With two-phase commit, if all non-failed participants are uncertain, they are blocked
- How to remove this uncertainty period to as large degree as possible?

# Three-phase commit

---

## Principle

- The protocol is designed based on the idea that *if any non-failed processor is uncertain then no processor (failed or non-failed) can have decided to commit*
- If all non-failed processors discover that they are all uncertain, they can then decide to abort
- When a failed processor recovers, it is told to abort too

# Three-phase commit

---

- ✍ Two-phase commit has blocking because the coordinator sends Commit to participants while they are uncertain
- ✍ The trick is for the coordinator to first remove participants from uncertain state, and when the coordinator knows that participants are no longer in uncertain state, then it sends the Commit
  - This adds one round of messages exchange

# Three-phase commit

---

## The idea

- When coordinator has received Yes from all, it sends PreCommit to all
- Each participant acknowledges the PreCommit
- When the coordinator has received all the Ack for the PreCommit, it learns that no participant is uncertain anymore. At this point, the coordinator sends Commit to participants

## Note:

- During the voting, if a processor votes No, the three-phase commit protocol behaves exactly as the two-phase commit protocol

# Three-phase commit: The algorithm (1/3)

---

## Coordinator

- Precommits the transaction (E)
- Sends VoteRequest to participants, and starts to collect Votes (U)
- If any of the votes was No or if the coordinator vote is No, then
  - Coordinator decides Abort and sends Abort to all participants (A)
- Otherwise, coordinator sends PreCommit to all participants, and starts to wait for Ack for the PreCommit (Ce)
- When a majority of Ack is received, it decides Commit and sends Commit to all participants (C)

# Three-phase commit: The algorithm (2/3)

---

## Participant p

- Initially waits for VoteRequest or Abort (E)
- Upon receipt of VoteRequest
  - If p can commit then
    - p precommits the transaction (U)
    - p sends Yes to coordinator
    - p starts to wait for PreCommit or Abort
  - Otherwise
    - p aborts the transaction (A)
    - p sends No to coordinator and stops

# Three-phase commit: The algorithm (3/3)

---

## Participant $p$ (in $U$ state)

- Upon receipt of PreCommit
  - $p$  sends Ack to coordinator (Ce)
  - $p$  starts to wait for Commit
- Upon receipt of Abort
  - $p$  aborts transaction and stops (A)

## Participant $p$ (in $C_e$ state)

- Upon receipt of Commit
  - $p$  commits the transaction (C)

# Three-phase commit: Handling failures

---

## Add timeout actions

- What a processor does when it does not receive an expected message

## Recovery protocol

- What a processor does when it recovers from a failure

# Three-phase commit: Timeout actions (1/2)

---

- ✍ **Coordinator timeout waiting for Votes**
  - Aborts and sends Abort
- ✍ **Participant timeout waiting for VoteRequest**
  - Abort and stops
- ✍ **Coordinator timeout waiting for Ack**
  - Decides Commit even if some processors have failed!
  - Is this fine?
  - Why?

# Three-phase commit: Timeout actions (2/2)

---

- ✍ Participant timeout waiting for PreCommit or Abort
  - Run a **termination protocol** to reach a consistent decision
  - Because at this point the participant is uncertain
- ✍ Participant timeout waiting for Commit
  - Run a **termination protocol** to reach a consistent decision
  - Because current coordinator may have failed after it sends PreCommit only to this participant

## Three-phase commit: Termination protocol (1/3)

---

- ✍ A new coordinator is elected
- ✍ The coordinator collects the state of all non-failed participants and performs the appropriate termination rule
- ✍ **Termination rules**
  - If some collected state indicates Abort, the new coordinator decides Abort and sends Abort to all participants and stops.

# Three-phase commit: Termination protocol (2/3)

---

## Termination rules (cont.)

- If some processor reports  $C$ , the new coordinator decides Commit and sends Commit to all participants and stops
- If at least one processor report  $C_e$  and the processors that reported  $U$  and  $C_e$  form the majority, then the new coordinator sends a PreCommit to all processors that did not report  $C_e$ . The coordinator decides Commit and sends Commit when a majority of processors have reported  $C_e$

## Three-phase commit: Termination protocol (3/3)

---

### Termination rules (cont.)

- If no processor reports  $C_e$ , and the majority reports  $U$  and  $A_e$ , then the new coordinator sends PreAbort to all processors that did not report  $A_e$  or  $C_e$ . The coordinator decides Abort and sends Abort when a majority of all processors have reported  $A_e$
- If none of the above rules applies, block until when a one of the above rules applies

# Replicated data management

---

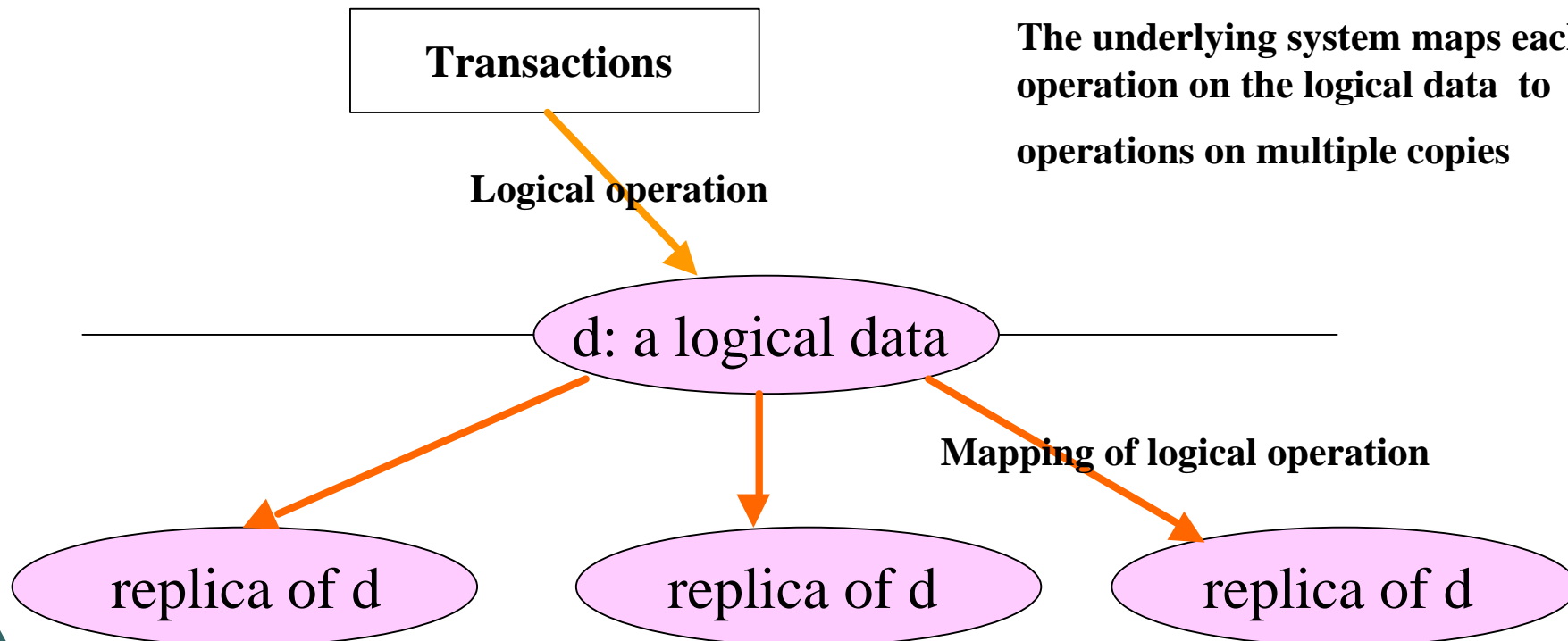
# Replicated data management

---

## Intuition

**Transactions see logical data**

**The underlying system maps each operation on the logical data to operations on multiple copies**



# Replicated data management: Correctness criteria

---

- ✍ Interleaved execution of concurrent transactions on replicated data is equivalent to a serial execution of those transactions on a one-copy database (one-copy serializable execution)
- ✍ Supporting concept
  - Serializability

## Serializability:

### Case of non-replicated database system

---

- ✍ Let  $T = \{T_0, T_1, \dots, T_m\}$  be a set of concurrent transactions
- ✍  $R_i$ : set of data items transaction  $T_i$  reads
- ✍  $W_i$ : set of data items transaction  $T_i$  writes
- ✍  $T_0$ : transaction that writes the initial values of the data items

Serializability:

Case of non-replicated database system

---

✍ During their execution transactions in **T**, will access overlapping sets of data and their data access will **conflict**

✍ **Conflicting operations?**

# Serializability:

## Case of non-replicated database system

---

### Conflicting operations

- Two operations are said to conflict
  - If they both access the same data and
  - At least one of these operations is a write
- If two operations conflict, their order of execution matters
  - Because, based on these conflicts one can determine how transactions must be ordered in an equivalent serial execution

# Serializability:

## Case of non-replicated database system

---

### Conflicting operations

- **Write-Write conflict**

- $T_i$  issues a write for  $d$  and current value of  $d$  was written by  $T_j$
- $T_i$  and  $T_j$  have *write-write* conflict denoted by  $T_j \text{ ? }_{ww} T_i$
- $T_j$  **must** occur before  $T_i$  in **any** serial execution of transactions in  $\mathbf{T}$

## Serializability: conflicting operations (cont.)

---

- ✍ Assume each data item passes through a sequence of versions due to update operations
- ✍ The initial value of data item  $d$  is  $d_0$
- ✍ If transaction  $T_i$  updates  $d_k$ , this results in  $d_{k+1}$  and we say that  $T_i$  writes (or produces)  $d_{k+1}$
- ✍ Given a version  $d_k$  of the data item  $d$ , we write  $T(d_k)$  to denote the transaction that wrote (produced)  $d_k$

# Serializability: conflicting operations (cont.)

---

## Write-read conflict

- If  $T_i$  reads version  $d_k$ ,  $k > 0$ , then  $T_i$  **must** occur after  $T(d_k) = T_j$  in **any** serial execution of transactions in  $T$
- This is a *write-read* conflict, denoted  $T_j \text{ ? }_{wr} T_i$

# Serializability: conflicting operations (cont.)

---

## Read-Write Conflict

- If  $T_i$  reads version  $d_k$ ,  $k > 0$ , then  $T_i$  **must** occur before  $T(d_{k+1}) = T_j$  in **any** serial execution of transactions in  $T$
- This is a *read-write* conflict, denoted  $T_j \text{ ? }_{rw} T_i$

# Serializability:

## Case of non-replicated database system

---

### Notation

- We write  $T_i ? T_j$  to say that one of the following holds
  - $T_i ?_{ww} T_j$
  - $T_i ?_{wr} T_j$
  - $T_i ?_{rw} T_j$

### Remark

- The outcome of a concurrent execution of transactions depends only on the relative ordering of conflicting operations


# Serializability:

## Case of non-replicated database system

---

### Rule for serialization

- If  $T_i \prec T_j$  then,  $T_i$  **must** occur before  $T_j$  in **any** serial execution of transactions in  $T$

 Constructing a serializable execution of concurrent transactions amounts to find an ordering of these transactions such that the above rule is respected

 Given an execution of concurrent transactions, how to determine if this execution is serializable?

- Serialization graph can help to answer this question

# Serializability:

## Case of non-replicated database system

---

### **Serialization Graph (SG)**

- A tool for determining a posteriori if an execution is serializable
- **SG(T, E)**: serialization graph for transactions in **T** for the execution **E**
- Nodes of **SG(T, E)** are transactions in **T**
- A directed edge  $(T_i, T_j)$  is in **SG(T, E)** iff  $T_i ? T_j$

Serializability:

Case of non-replicated database system

---

## **Central Theorem of Serializability**

- If  **$SG(T, E)$**  is acyclic (i.e. has no cycles), then the execution  **$E$**  of transactions in  **$T$**  is serializable


# Serializability:

## Case of non-replicated database system

---

### **Serialization Graph illustrated**

- $T = \{T_1, T_2, T_3\}$
- $T_1 :: R_1(x) W_1(z)$
- $T_2 :: W_2(x) W_2(y)$
- $T_3 :: R_3(y) W_3(z)$

  $E = R_1(x) W_2(x) W_2(y) R_3(y) W_3(z) W_1(z)$

- (incomplete execution: no commit/abort)

  $SG(T, E)$  is  $T_1 ?_{rw} T_2 ?_{wr} T_3 ?_{ww} T_1$

- There is a cycle, therefore  $E$  is not serializable

# Serializability:

## Case of non-replicated database system

---

- ✍ Well, serialization graph can be used to say whether a given execution is serializable
- ✍ In general, we want the database system to produce serializable executions (recall the role of SCH)
- ✍ Serializable executions can be produced using
  - **Two-phase locking**
  - Timestamp ordering
  - Etc.

# Serializability:

## Two-Phase Locking (2PL) (1/5)

---

- ✍ Each data item is protected by a lock
- ✍ A lock is either
  - A “read lock” or
  - A “write lock”
- ✍ A transaction must
  - Obtain a read or write lock on data item d before reading d
  - Obtain a write lock on d before updating d

# Serializability:

## Two-Phase Locking (2PL) (2/5)

---

- ✍ Several transactions can have “read lock” on the same data item
- ✍ If a transaction has a “write lock” on x, no other transaction can obtain a “read lock” or “write lock” on x

# Serializability: Two-Phase Locking (2PL) (3/5)

---

✍ Let

- $x$  be a data item that is currently locked
- $\text{LockReq}(x)$  be a request for locking  $x$

✍ We say that  $x$  is locked in a conflicting mode with respect to  $\text{LockReq}(x)$  if one of the following holds

- $\text{LockReq}(x)$  is a “read lock” request while  $x$  is locked on write mode
- $\text{LockReq}(x)$  is a request for a “write lock” while  $x$  is locked on read mode
- $\text{LockReq}(x)$  is a request for a “write lock” while  $x$  is locked on write mode

# Serializability:

## Two-Phase Locking (2PL) (4/5)

---

✍ A scheduler implementing the two-phase locking strategy for concurrency control has mainly two rules

### ✍ Rule 1

- When it receives a LockReq(x) from a transaction T
  - If x is already locked in “conflicting mode”, then T is forced to wait until it can obtain the lock needed
  - Otherwise, x is locked on the mode of the request, and the lock is granted to T

# Serializability: Two-Phase Locking (2PL) (5/5)

---

## Rule 2

- Once SCH has released lock for a transaction T, SCH may not subsequently grant any new locks (on any data item) for T
  - This ensures that all pairs of conflicting operations of two transactions are scheduled in the same order

## Note

- 2PL can lead to deadlock
- You will learn more from books on database

# Basic approaches to RDMS

---

# Basic approaches to RDMS: "Write-all" approach (1/2)

---

## ✍ Write-all approach

- When the RDMS receives a  $\text{Read}(x)$  from a transaction
  - This  $\text{Read}(x)$  is translated into  $\text{Read}(x_A)$  where  $x_A$  is **any** copy of item  $x$
- When the RDMS receives a  $\text{Write}(x)$  from a transaction
  - This  $\text{Write}(x)$  is translated into  $\{\text{Write}(x_{A1}), \text{Write}(x_{A2}), \dots, \text{Write}(x_{Am})\}$  where  $\{x_{A1}, \dots, x_{Am}\}$  are **all** copies of  $x$
- Uses any concurrency control algorithm to synchronize access to copies

# Basic approaches to RDMS: “Write-all” approach (2/2)

---

## ✍ Properties of “Write-all” approach

- If no failure, then everything is fine
- (-) If nodes can fail and recover, then the write-all approach suffers from the *availability problem*
  - If some nodes storing x are down, then RDMS must delay processing of any Write(x) until when all copies can be written
  - The all poses reduces availability, the RDMS must write to **all** copies, even if some nodes have failed !

# Basic approaches to RDMS:

## “Write-all-availabe” approach (1/2)

---

### Write-all-available approach

- When the RDMS receives a Read( $x$ ) from a transaction
  - This is translated into Read( $x_A$ ) where  $x_A$  is **any** copy of item  $x$
- When the RDMS receives a Write( $x$ ) from a transaction
  - This is translated into  $\{\text{Write}(x_{A1}), \dots, \text{Write}(x_{Av})\}$  where  $\{x_{A1}, \dots, x_{Av}\}$  are **all available** copies of  $x$
- Uses any concurrency control algorithm to synchronize access to copies

## Basic approaches to RDMS: "Write-all-available" approach (2/2)

---

### Properties of the Write-all-available

- (+) solves the availability problem
- (-) introduces the correctness problem
  - Copies not available during the last update become out-of-date
  - Use of out-of-date copies can lead to non-one copy serializable executions