CES-27 & CE-288
Distributed Programming

Chapter 5 – Distributed transactions

Celso Massaki HIRATA

hirata@ita.br

Instituto Tecnológico de Aeronáutica

Agenda

1. Introduction
- 1.1. Distributed databases
- 1.2. Management of distributed transactions
- 1.3. Techniques
  - 1.3.1. Two-phase-commitment for recovery
  - 1.3.2. Two-phase-locking for concurrency control

2. A framework for transaction management
- 2.1.Properties of transactions
- 2.2.Distributed transactions

3. Supporting atomicity of distributed transactions
- 3.1 Recovery in centralized systems
  - 3.1.1. A model of failures in centralized databases
  - 3.1.2. Logs
  - 3.1.3. Recovery procedures
- 3.2 Recovery of distributed transactions
  - 3.2.1. The 2-phase-commitment protocol protocol
  - 3.2.2.. Some comments on the 2-Phase-commitment

4. Concurrency control for distributed transactions
- 4.1. Concurrency control based on locking in centralized databases
- 4.2. Concurrency control based on locking in distributed databases
- 4.3. Some comments on distributed 2-phase locking
  - 4.3.1. Two phase locking and availability
  - 4.3.2. Two phase-locking and recovery

1. Introduction

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Distributed databases

- **Collection of data** which are **distributed** over different **computers** of a computer **network**.

- Each **site** of the **network**:
  - ✓ Has **autonomous processing** capability and can perform **local applications**.
  - ✓ Participates in the **execution** of at least one **global application** which requires accessing **data** at several **sites** using a **communication subsystem**.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Definition of transaction

- A **transaction** is an **atomic unit** of **database access**, which is **either completely executed** or **not executed** at all.
  - ✓ **Example**
    - Transferring **$1000** from an **account *A*** to an **account *B***

- The notion of **transaction** is therefore **related** to the problems of **recovery** and **concurrency control**
  - ✓ **Atomicity** must be preserved both in case of **failures** and in case of **concurrent execution**.

→ 1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Management of distributed transactions

- The **management** of **distributed transactions** requires dealing with several **problems** which are strictly **interconnected**
  - ✓ **Reliability**
  - ✓ **Concurrency control**
  - ✓ Efficient utilization of **resources**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Techniques

- Techniques used in this **chapter** to understand the basic **aspects** os **distributed transaction management**:
  - ✓ **2-phase-commitment** for **recovery**
  - ✓ **2-phase-locking** for **concurrency control**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

2. A framework for transaction management

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

A framework for transaction management

- In this **section** we will:
  - ✓ Define the **properties** of **transactions**
  - ✓ Discuss the **goals** of **distributed transaction management**
  - ✓ Present a **model** of a **distributed transaction**

Properties of transactions

- A **transaction** is an **application** or **part** of an **application** which is characterized by the following **properties**:
  - ✓ **A**tomicity
  - ✓ **C**oncurrency control
  - ✓ **I**solation
  - ✓ **D**urability

1. Introduction
→ 2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Atomicity

- Either **all** or **none** of the **transaction's operations** are **performed**.
  - ✓ Transfer between two banking accounts has two operations: **debit** and **credit**
- Requires that if a **transaction** is interrupted by a **failure**, its **partial results** are undone.
- There are two reasons why a **transaction** is not completed
  - ✓ Transaction **aborts**
  - ✓ System **crashes**

Atomicity

- The **abort** of a **transaction** can be requested by the **transaction** itself, the **user** or by the **system**

- The **activity** of **ensuring atomicity** in the presence of **transaction aborts** is called **transaction recovery**

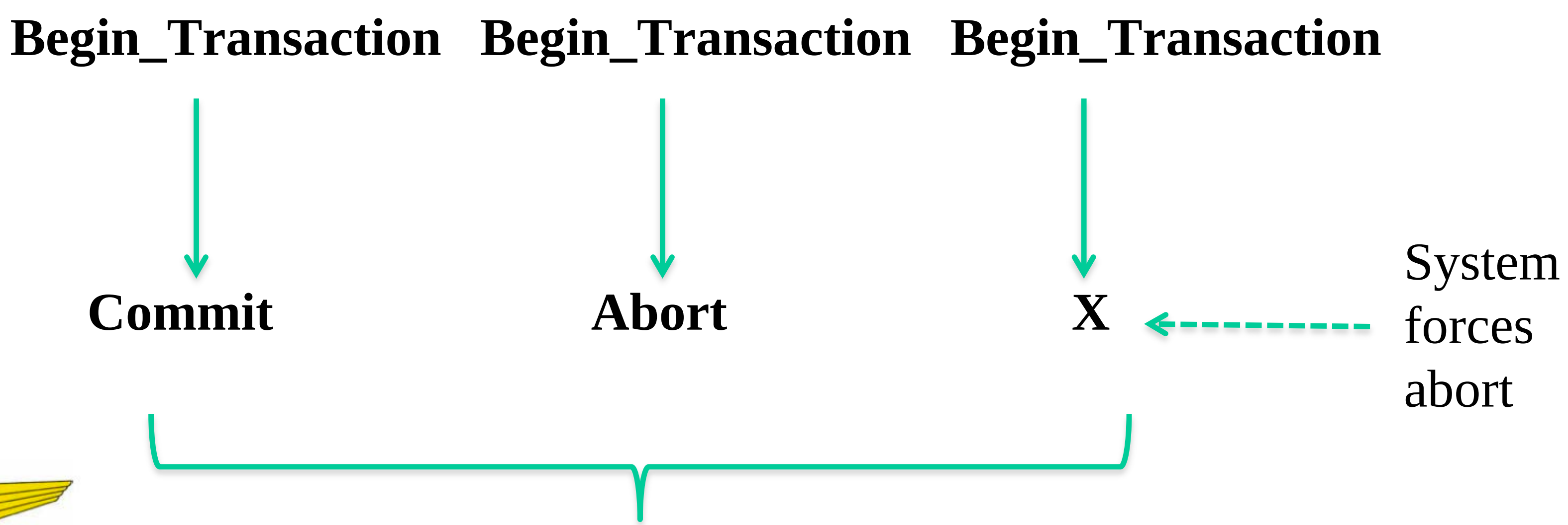
- The **activity** of **ensuring atomicity** in the presence of **system crashes** is called **crash recovery**

1. Introduction
→ 2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Atomicity

- The **completion** of a transaction is called **commitment**

- Each transaction begins with a ***begin_transaction*** primitive and ends with an ***abort*** or ***commit*** primitives



Types of transaction termination

Properties of transactions

- **Durability**:
  - ✓ Once a transaction has committed, the system must guarantee that the results of its operations will never be lost
    - Independent of subsequent failures
    - Database recovery is the activity of providing the transactions durability

1. Introduction
→ 2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Properties of transactions

- **Serializability**
  - If several **transactions** are executed **concurrently**, the **results** must be the **same** as if they were **executed serially** in some order.
    - **Serializability** is **correct by definition**!
    - **Concurrency control** is the activity of guaranteeing transaction's serializability.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Properties of transactions

- Example of **concurrency control** with **two transactions**:
  - ✓ ***T1:***
    - Deposit of **R$ 1,00** in account ***A***
    - Initial balance, **R$ 49,999.00**
  - ✓ ***T2:*** Interest payment on ***A***
    - **1%** if **Value >= R$ 50,000.00**
    - **0,5%** otherwise

Properties of transactions

- Example of **concurrency control** with **two transactions (cont.)**:
  - ✓ Operations of ***T1***:
    - $R_1(A)$, $W_1(A)$
  - ✓ Operations of ***T2:***
    - $R_2(A)$, $W_2(A)$

1. Introduction
→ 2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Properties of transactions

- Example of **concurrency control** with **two transactions (cont.)**:
  - ✓ If the sequence is ***T1*** and ***T2***, A=50.500,00
  - ✓ If the sequence is ***T2*** and ***T1***,
    - A=50.249,99 (50.248,995+1)
    - Both sequences are correct.
  - ✓ If the operations of ***T1*** and **T2** are executed in a intercalated (interleaved) way:
    - $R_1(A)$, $R_2(A)$, $W_1(A)$, $W_2(A)$
    - A=50.248,99!
    - Inconsistent

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Properties of transactions

- **Isolation** (visibility)
  - An **incomplete transaction** cannot reveal its results to other **transactions** before commitment.
  - This **property** is needed in order to avoid the problem of **cascading aborts**
    - **Domino effect**

Properties of transactions

- **Isolation** (visibility)

T1: Credita V para A (A inicialm. 0)		T2: Debita W de A
Begin T1		
…		Begin T2
A:=A+V		…
…		If A>W then
	A=V se sem isolação	A:= A-W
…		else abort
…		…
Falha		Commit T2

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Properties of transactions

- **Isolation** (visibility)
  - ✓ Suppose that initially, ***A*** is **0** and ***V*** > ***W***
  - ✓ What happens?
    - **Problem**: If there is no isolation ***T2*** commits even in case of ***T1*** failure.
    - ***T2*** cannot be undone

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

3. Supporting atomicity of distributed transactions

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Supporting atomicity of distributed transactions

- In this **section**:
  - ✓ We will discuss the **atomicity** of **distributed transactions**
  - ✓ We will discuss the importance of **logs** for **recovery**
  - ✓ We will present the **2-phase-commitment protocol**
  - ✓ We will also derive a **reference model** of a **distribute transaction manager**
    - Which can be used for describing **recovery algorithms**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery in centralized systems

- **Recovery mechanisms** are built by allowing the resumption (retomada) of **normal operations** of a **database** after a **failure**.
  - ✓ Therefore, before studying **recovery**, we must analyze the **failures types** that can happen in a **centralized database.**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

A model of failures in centralized databases

1. Failures without **loss of information**.
   - ✓ Example: division by zero.
2. Failures with **loss of volatile storage**.
   - ✓ Example:: system crashes, power outage
3. Failures with **loss of nonvolatile storage**
   - ✓ Example: media failures, head crashes

1. Introduction
2. A framework for transaction management
→ 3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

A model of failures in centralized databases

- The **probability of failures** with **loss of nonvolatile storage** is less than the other two types.
  - ✓ It can be reduced implementing **stable storage**.
  - ✓ It consists in **replicating** the information at two or more disks with **independent failure mode** and make a careful update
    - Each **copy** is **updated** and **verified** in turn

Logs

- Stored in nonvolatile storage (disk)
- A **log** contains information for **undoing** or **redoing** all **actions** which are performed by **transactions**.
- The **undo** e **redo** operations must be **idempotent**
  - ✓ i.e., performing them several times should be equivalent to performing them once

    UNDO(UNDO(UNDO(…(action)…)))) = UNDO (action);

    REDO(REDO(REDO(…(action)…)))) = REDO (action);

Logs

- **Idempontent operations** are necessary because the **recovery procedure might** fail and be restarted several times
  - ✓ It is also very convenient since we are relieved from the need of knowing whether an **action** that we want to **undo** or **redo** was already **undone** or **redone**.

Log record

1. The **identifier** of the **transaction**
2. The **identifier** of the **record**
3. The **type** of **action** (insert, delete, modify)
4. The **old** record **value** (required for the UNDO)
5. The **new** record **value** (required for the REDO)
6. Auxiliary information for the **recovery procedure**
   - ✓ Typically, a pointer to the previous log record of the same transaction

Log record example

- Transfer **$100** from account **A** to account **B**
  - ✓ Initially **A=1000**, **B=800**.
- Entries in the **log record**

1. <T1, begin-trans>
2. <T1, A, modify, 1000, 900>
3. <T1, B, modify, 800, 900>
4. <T1, commit>

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Log record

- The writing of a **database update** and the writing of the corresponding **log record** are two distinct **operations**
  - ✓ If the **database update** were performed before writing to the **log record**, the **recovery procedure** would be unable of undoing the **update**.
  - ✓ In order to avoid this problem, the **log write-ahead protocol** is used.

“Log write-ahead” protocol

- The **log write-ahead protocol** has **two** basic **rules**:

1. Before performing a **database update**, at least the **undo** portion of the corresponding **log record** must have been already **recorded** on **stable storage**.
2. Before **committing a transaction**, all **log records** of the **transactions** must have already been **recorded** on **stable storage**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery procedures

- When a **failure** with **loss of volatile storage** occurs, a **recovery procedure** reads the **log file** and performs the following **operations**:
  1. Determine all **non-committed transactions** that have to be **undone**
  2. Determine all **transactions** which need to be **redone**
  3. Undo the **transactions** determined at **step 1** and **redo** the **transactions** determined at **step 2**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery procedures

- **Checkpoints** are operations which are periodically performed in order to make more efficient the first **two steps** of the **recovery procedure**.
  - A**voids total recovery**
  - We periodically record **checkpoint records** in the **log file**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery procedures

➢ Recovery with **checkpoint** requires the following **operations**:

1. Writing to **stable storage** all **log records** and all **database updates** which are still in **volatile storage.**
2. Writing to **stable storage** a **checkpoint record**:
   - Set of **active transactions** that has not been **committed** or **aborted**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery procedure using checkpoints

- Using **checkpoints**, **steps 1** and **2** of the **recovery procedure** are substituted by the following:
  1. Find and read the **last checkpoint record**
  2. Put all **transactions** written in the **checkpoint record** into the **undo set**. The **redo set** is initially empty
  3. Read the **log file** starting from the **checkpoint record** until its end
     - If a ***begin_transaction*** record is found, put the corresponding **transaction** in the **undo set**
     - If a ***commi***t record is found, move the corresponding **transaction from** the **undo set** to the **redo set**.

Recovery procedure using checkpoints

- **Example**:
  - ✓ Suppose the **checkpoint record** is recorded from **10** to **10** minutes starting at **24 hours**
  - ✓ Suppose the **system collapses** at **11h09m** and the **log file** has the following content (**next slide**)

1. Introduction
2. A framework for transaction management
→ 3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery procedure using checkpoints

11h00m	<T5, T8, T10>	3 **active transactions**
11h01m	<T12, **begin-trans**>	**T12** is activated
11h02m	<T8, A, **modify**, 1000, 900>	
11h03m	<T10, **commit**>	**T10** completes
11h04m	<T13, **begin-trans**>	**T13** is activated
11h05m	<T13, D, **modify**, 5000, 200>	
11h06m	<T13, **commit**>	**T13** completes
11h07m	<T12, C, **modify**, 110, 145>	

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery procedure using checkpoints

11h00m	<T5, T8, T10>	U={T5, T8, T10} R={}
11h01m	<T12, **begin-trans**>	U={T5, T8, T10,**T12**} R={}
11h02m	<T8, A, **modify**, 1000, 900>	
11h03m	<T10, **commit**>	U={T5, T8,T12} R={**T10**}
11h04m	<T13, **begin-trans**>	U={T5, T8,T12,T13} R={**T10**}
11h05m	<T13, D, **modify**, 5000, 200>	
11h06m	<T13, **commit**>	U={T5, T8,T12} R={T10,T13}
11h07m	<T12, C, **modify**, 110, 145>	

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery procedure using checkpoints

- Conceptually a **log** contains the **whole history** of the **database**.
  - ✓ However, only the **latest portion** refers to **transactions** which might be **undone** or **redone**.
  - ✓ Therefore, only this **latest portion** of the **log** must be **kept online**
    - While the **remainder** of the **log** can be kept in **offline storage** (tape).

Failures with loss of stable storage

- So far, we have considered only the **recovery from failures without loss of stable storage**.
  - ✓ **Recovery techniques** have been developed for **loss of stable storage** as well.

- It is important to distinguish **two possibilities** when talking about **stable storage**:
  1. Failures in which **database** information is **lost**, but **logs** are **safe**
  2. Failures in which **log** information is **lost**

Failures with loss of stable storage

- In the **first case** the **recovery technique** consists of performing a redo of all **committed transactions** using the log.
  - ✓ The **redo** is performed after having reset the database to a **dump**
    - An image of a **previous state** which was stored on **offline storage**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Failures with loss of stable storage

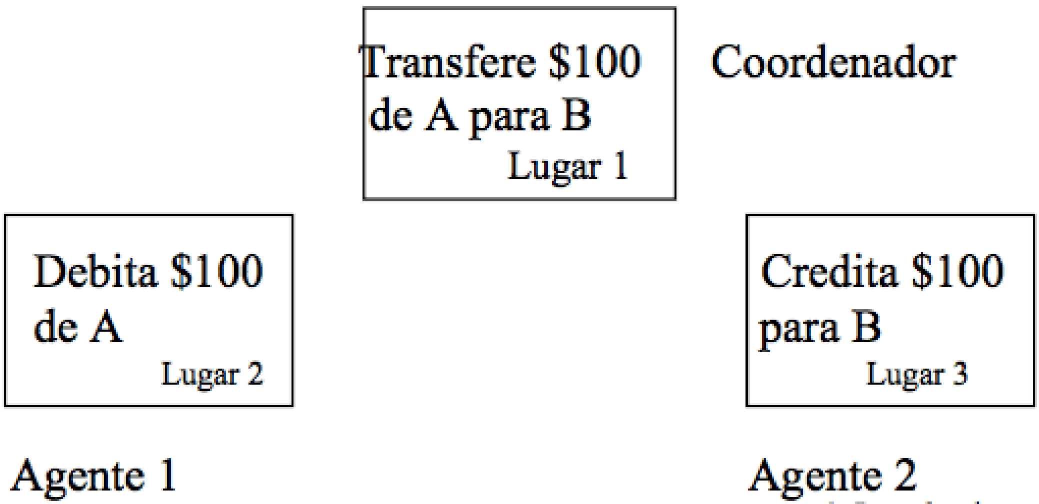
- In the **second case** it is in general **impossible** to completely **restore** the most recent **database state.**
  - ✓ Violating **database durability**
  - ✓ **Catastrophic** event that **should never happen**!

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

2-Phase commitment protocol: recovery of distributed transactions

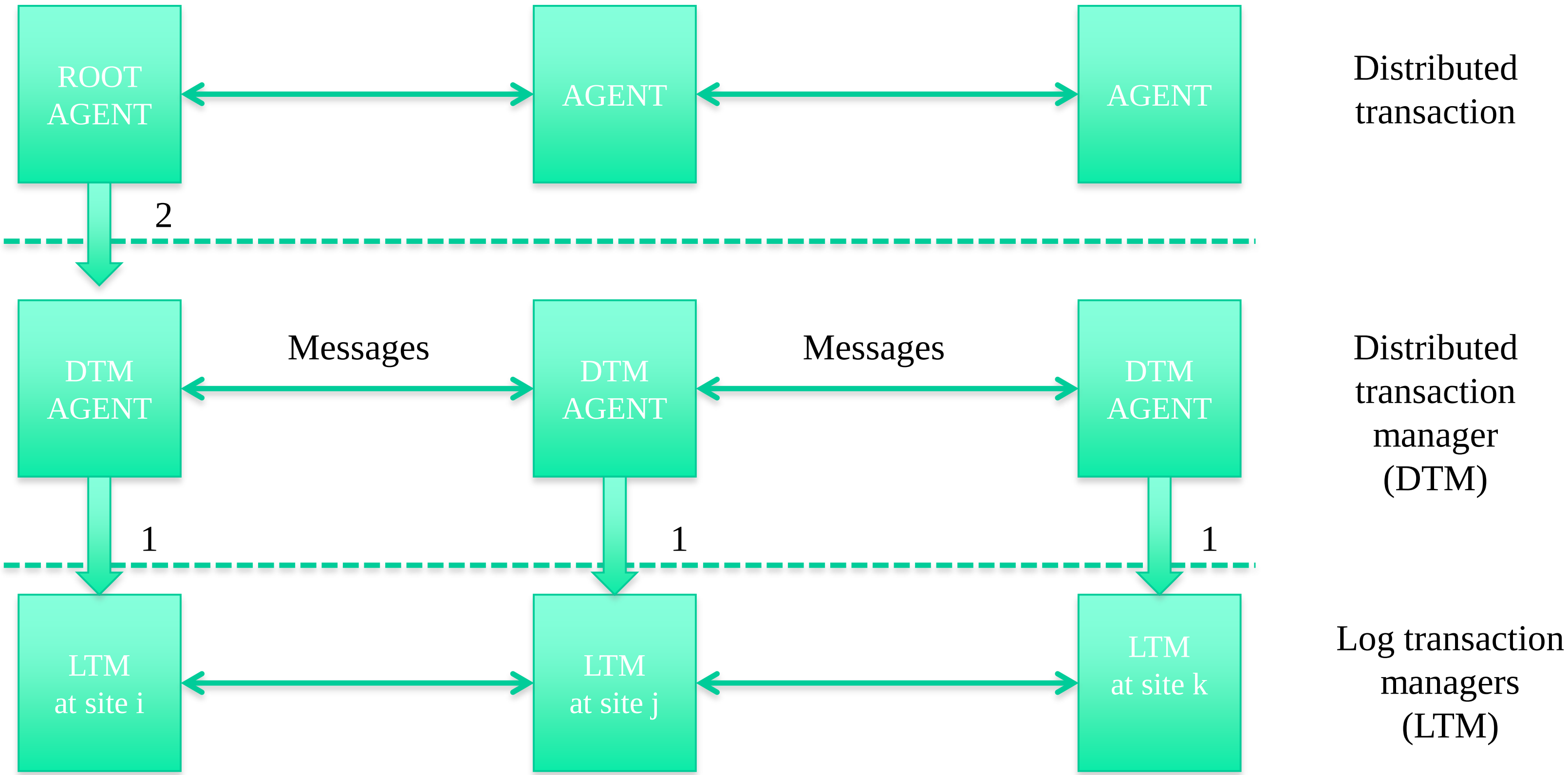
2-Phase commitment protocol: recovery of distributed transactions

- Deal with **failures** at different **sites**
- Example of a **distributed transaction**



1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Reference model for implementation of distributed transaction recovery



Interface 1: Local_Begin, Local_Commit, Local_Abort, Local_Create
Interface 2: Begin_Transaction, Commit, Abort, Create

Recovery of distributed transactions

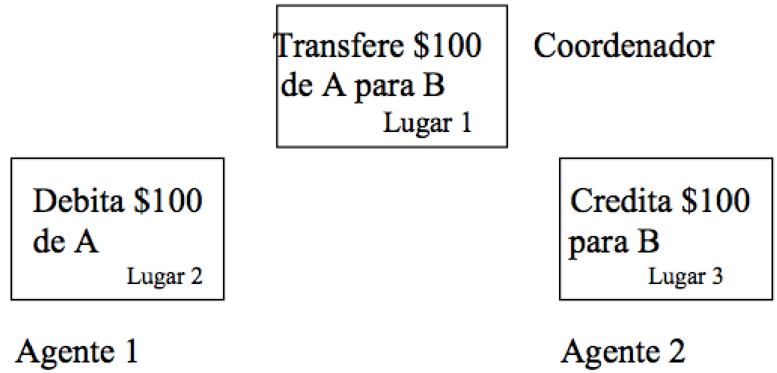
- Each **agent** allows **local centralized recovery services**
  - ✓ **Stable memory** and **transactions log**.

- The **coordinator** of a **distributed transaction** can be co-located with one of the **agents** and use the log of that **agent** or the another **log**.

Recovery of distributed transactions

- The **coordinator** communicates with the **agents** by **message passing** as follows:

- Begin of **transaction**:
  - ✓ **Coordinator**
    - write **global-begin** to log;
    - send **begin-trans** to agents;



- **Agents**:
  - ✓ receive **begin-trans** from coordinator;
  - ✓ write **local-begin** to log;

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery of distributed transactions

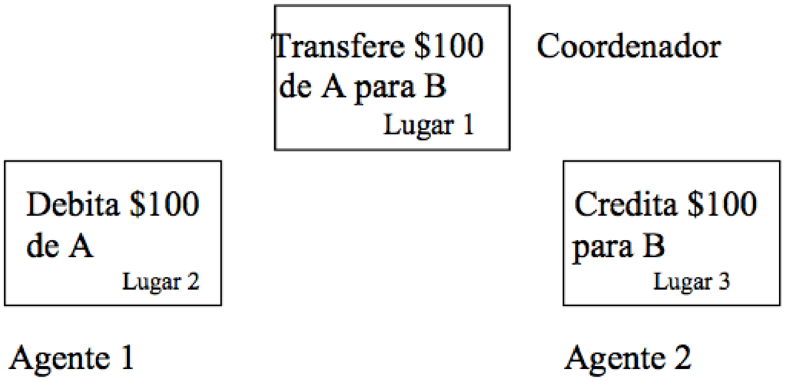
- Effectively, **agents** span local **sub-transactions** of the **global transaction**.

- All subsequent **messages** are **timestamped** with the **transaction identifier**.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

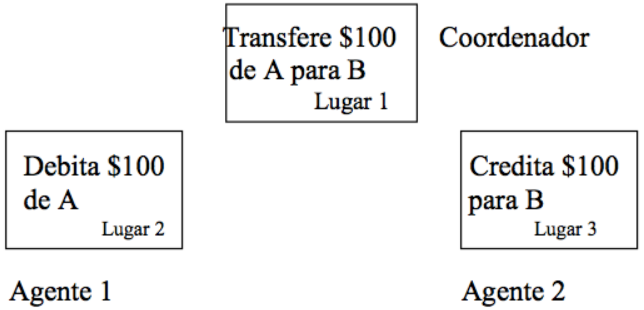
Recovery of distributed transactions

- Application **messages** in the example:
  - ✓ **Coordinator**:
    - Send debit \$100 from A to **Agent 1**
    - Send credit \$100 to B to **Agent 2**



Recovery of distributed transactions

- **Agent 1**: (initially *A= 1000*)
  - ✓ Receive from **coordinator**
  - ✓ Write <Tid, A, modify, 1000, 900> to log
  - ✓ Update A



- **Agent 2**: (initially *B= 800*)
  - ✓ Receive from **coordinator**
  - ✓ Write <Tid, B, modify, 800, 900> to log
  - ✓ Update B

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery of distributed transactions

- The **agents** do not need to make permanent changes in the **log** or in the **database** until receive a **commit command**.
  - ✓ This optimizes the abort where temporary changes can be discarded.

1. Introduction
2. A framework for transaction management
→ 3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Recovery of distributed transactions

- **Abort**:
  - ✓ **Coordinator**:
    - Write **global-abort** to log;
    - Send abort to agents;

- **Agents**:
  - ✓ Receive abort from **coordinator**;
  - ✓ Write **local-abort** to log;
  - ✓ Undo actions up to **local-begin**

Recovery of distributed transactions

- **Commit**
  - The **implementation** of the commit primitive is the most **difficult** and expensive by the fact that the correct **commitment** of a **distributed transaction** requires that all its **sub-transactions commit locally**.
    - Even in the case of failures.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Phase one

- **Coordinator**:
  - ✓ Write “**PREPARE**” record in the **log**;
  - ✓ Send “**PREPARE” message** and activate **timeout**

Phase one

- **Participant**:
  - ✓ The **goal** is to obtain a **decision**
  - ✓ Wait for **PREPARE message**;
  - ✓ **If** participant is willing to commit then
    - Write **sub-transactions** record in the log;
    - Write "**READY**" record in the **log**;
    - Send **READY** answer message to **coordinator**;
  - ✓ **else**
    - Write "**ABORT**" record in the log;
    - send **ABORT** answer message to **coordinator**;

Phase two

- **Coordinator**:
  - ✓ Receive **READY** or **ABORT** from all agents
    - Or **timeout**
  - ✓ if **ABORT** received (or **timeout**) then
    - Write **global-abort** to log;
    - Send **abort** to agents;
  - ✓ else { all agents reply ready }
    - Write **global-commit** to **log**;
    - Send **commit** to agents;

Phase two

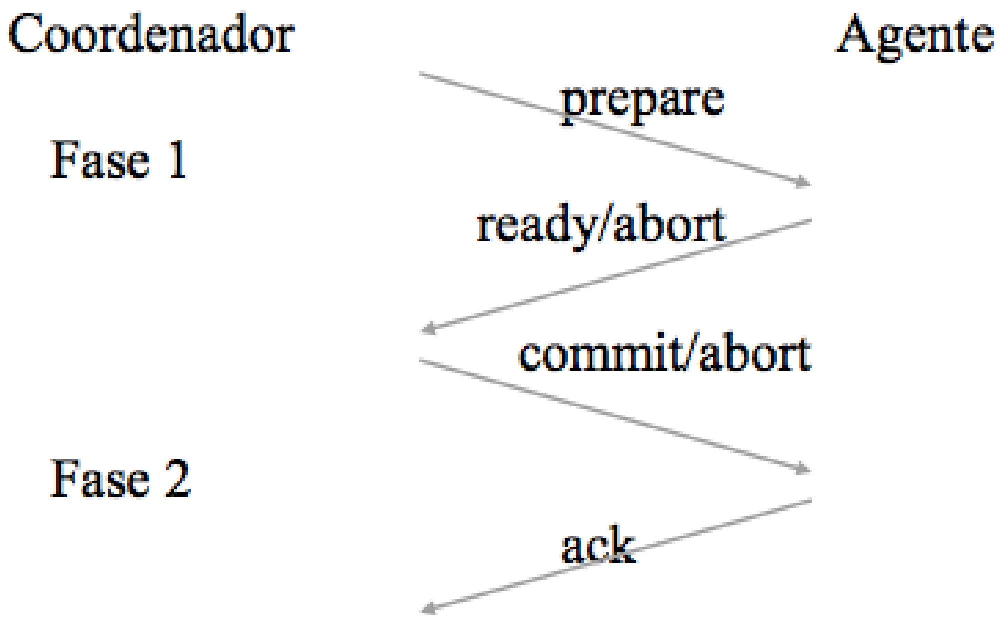
- **Agents**: Implements the decision
  - Receive **ABORT** or **COMMIT** from **coordinator**;
  - Write to **log**;
  - Send **ack** to coordinator;

- **Coordinator**:
  - Wait for **ACK** messages from all participants;
  - Write “**complete**” record in the log;

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Sequence diagram

- **One-to-many** communication **pattern**



1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Example of log stages

- Example of **log stages** after a **successful commit**
- **Coordinator**:

  <global-begin>

  <prepare, site2, site3> {records agents sites}

  <global-commit>

  <complete>

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Example of log stages

- **Agent 1**

  <local-begin>

  <modify, A, 1000, 900>

  <ready, site1> {records coordinator site}

  <local-commit>

- **Agent 2**

  <local-begin>

  <modify, B, 800, 900>

  <ready, site1> {records coordinator site}

  <local-commit>

Failure analysis

- The **2-phase commitment protocol** is **resilient** to all **failures** in which **no log information** is **lost**

- In the **next slides** we analyze the behavior of the protocol in the presence of **different kinds of failures**:
  1. **Site** failures
  2. **Message** loss
  3. **Network** partitions

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Site failures

a) ***A participant fails before having written the ready record in the log.***

- ✓ The **coordinator's timeout expires** and it takes the **abort** decision.
- ✓ All **operational participants abort** their sub-transactions.
- ✓ When the **failed participant recovers**, the **restart procedure** simply **aborts** the **transaction**, without having to collect information from other **sites**.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Site failures

b) A participant fails after having written the ready record in the log

- ✓ The **operational sites** correctly **terminate** the **transaction** (commit or abort).
- ✓ At restart, the **failure agent** has to ask the **coordinator** or some other **participant** about the **outcome** of the **transaction**
  - And then perform the appropriate action (**commit** or **abort**)
- ✓ What does it happen if the **agents fail after** the **log** has been written **ready** and before the **message** is sent?
  - Case **a** in the **previous slide**

Site failures

c) The coordinator fails after having written the prepare record in the log, but before having written a global_commit or global_abort record in the log.

- ✓ All participants which have already answered **READY** must wait for the recovery of the **coordinator**.
- ✓ At restart, the **coordinator** resumes the **commitment protocol** from the beginning (i.e. do prepare).
- ✓ Each ready participant must recognize that the new **PREPARE** message is a repetition of the previous one

Site failures

d) The coordinator fails after having written a global_commit or global_abort record in the log, but before having written the complete record in the log

- ✓ The **coordinator** at restart must send to all participants the decision again repeating the **phase 2** (sending the abort or commit decision
  - All participants which have not received the command have to wait until the coordinator recovers

Site failures

e) The coordinator fails after having written the complet record in the log

- ✓ **No action** is required **at restart**

Message loss

a) ***An answer message (READY or ABORT) from a participant is lost***

- ✓ Coordinator's **timeout expires** and the whole transaction is **aborted**

b) A PREPARE message is lost

- ✓ The **participant** remains in wait.
- ✓ The **coordinator** does not receive an answer.

1. Introduction
2. A framework for transaction management
→ 3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Message loss

c) A command message (COMMIT or ABORT) is lost

- ✓ The **destination participant** remains uncertain about the decision.
- ✓ The **problem** is eliminated introducing a **timeout** in the **participant**.
- ✓ If **no command** has been received after the **timeout interval** a request for repetition of the command is sent.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Message loss

d) An ACK message is lost

- ✓ The **coordinator** remains **uncertain** about the fact that the **participant** has received the **command message**.
- ✓ Problem eliminated with a **timeout** in the **coordinator**
- ✓ If no **ACK message** is received after the **timeout interval** the **coordinator** will send the **command** again.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Network partitions

- Let us suppose that a **simple partition** occurs, dividing the sites in two groups
  - ✓ The group which contains the coordinator is called the **coordinator-group**
  - ✓ The another group is called **participant-group**

A = coordinator + some agents

B = remainder agents

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Network partitions

- The **coordinator** sees **multiple failures** of **agents** on **B**
  - ✓ Cases **a** and **b** of host failure

- **Agents** on **B** see **coordinator failure**
  - ✓ Cases **c** and **d** of host failure

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Network partitions: notes

- Before an **agent** has recorded a **READY** entry in the **log**, it can in an autonomously way **abort** its **local sub-transactions.**

- After **READY**, the **agent** can remain **blocked** if the **coordinator** fails.

Network partitions: notes

- The **agent** must ensure all **resources** of the **sub-transaction** until the restart of the **coordinator** (e.g. locks)
  - since it does not know if the transaction will be **committed** or **aborted**
  - This reduces the **system availability**.

Network partitions: notes

- The finish is possible only if one of the participants had received the command (**ABORT/COMMIT**) or if neither **participant** has received the command or only the **coordinator** has failed.

- The finish is impossible when neither **operational participant** has received the command and the **coordinator** and **one** of the **participants** has **failed**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

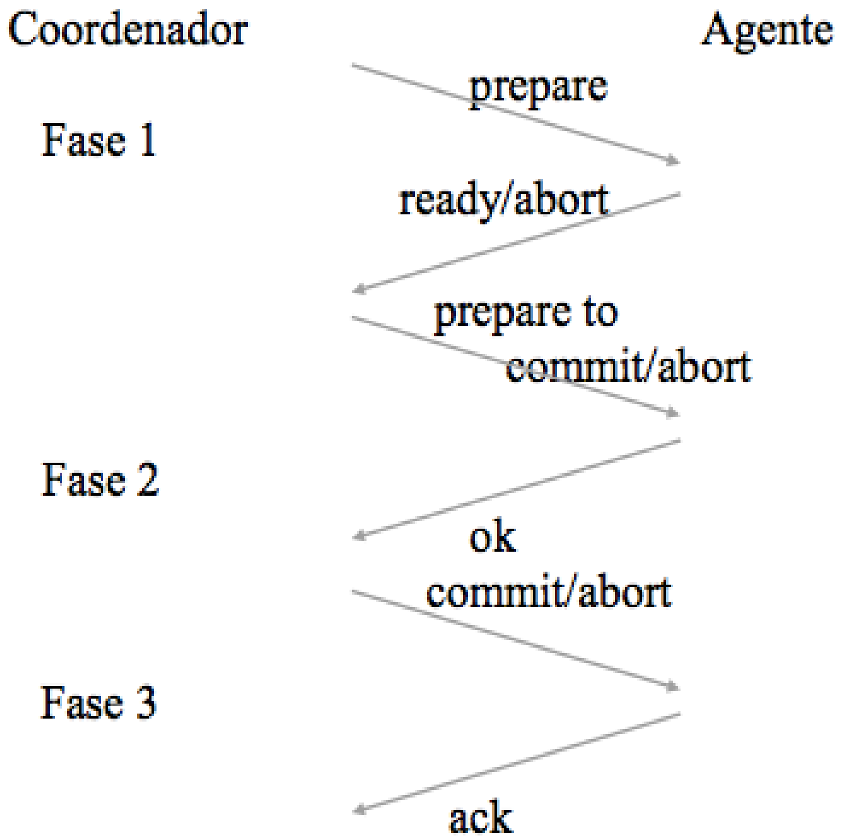
3-phase commitment protocol: addressing coordinator failure

- Used in case of **coordinator failure** in the **2-Phase protocol**
  - It is a **non-blocking commitment protocol**

- Within **2-Phase protocol**, if the **coordinator fails** after recording **PREPARE**, but before a **GLOBAL-COMMIT**, the agents that have recorded **READY** must wait for the **recovery** of the **coordinator** (**blocking protocol**).
  - But, if the **coordinator** does not return?

1. Introduction
2. A framework for transaction management
→ 3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

3-phase commitment protocol: addressing coordinator failure



1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

3-phase commitment protocol: addressing coordinator failure

- After the **READY**, the **agents** do not go directly to the **COMMITED** state
  - ✓ They go to the **PREPARE-TO-COMMIT** state

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

3-phase commitment protocol: addressing coordinator failure

- Eliminates the **BLOCKED state** of the **agents**:
  - If no agents have received the message **PREPARE-TO-COMMI**T (see **BLOCKED** state for the **2-phase protocol**) the **operational agents** can abort the transaction
    - Because the **failure agents** have still not **committed** yet.
    - **Failure agents abort** at **restart**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

3-phase commitment protocol: addressing coordinator failure

- A **finish protocol** is required in order to **elect** a **new coordinator** and complete the **transaction** if the **coordinator fails**.

- The **protocol** must **elect** a **new coordinator** and can be **centralized**.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Termination protocol

- Based on the **two properties** of the **non-blocking commitment protocol**:

1. If at least **one agent** did not enter the state **PREPARE-TO-COMMIT** then the transaction can be aborted (i.e. **no agent** has received the **COMMIT** message)

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Termination protocol

- Based on the **two properties** of the **non-blocking commitment protocol (cont.)**:

2. If at least **one operational agent** has reached the state **PREPARE-TO-COMMIT** then the **transaction** can be committed
    - ✓ i.e. **all agents** have previously answered **READY**.

Termination protocol

- Notice that the **cases 1** and **2** are **not mutually exclusive**.

- The **protocol** that **always commit** the **transaction** when both cases are possible is called **progressive protocol**

Termination protocol

- The **termination protocol** must **elect** a new **coordinator**.
  - This **election** can be **centralized** or **decentralized**.
  - Generally, we have **centralized** and **non-progressive termination protocol**.

Termination protocol

- The **non-blocking commitment protocol** is **catastrophic** for the **network partition case** since two groups can **elect** a **new coordinator** and obtain different decisions by accounting the **transaction** at the **two partitions**

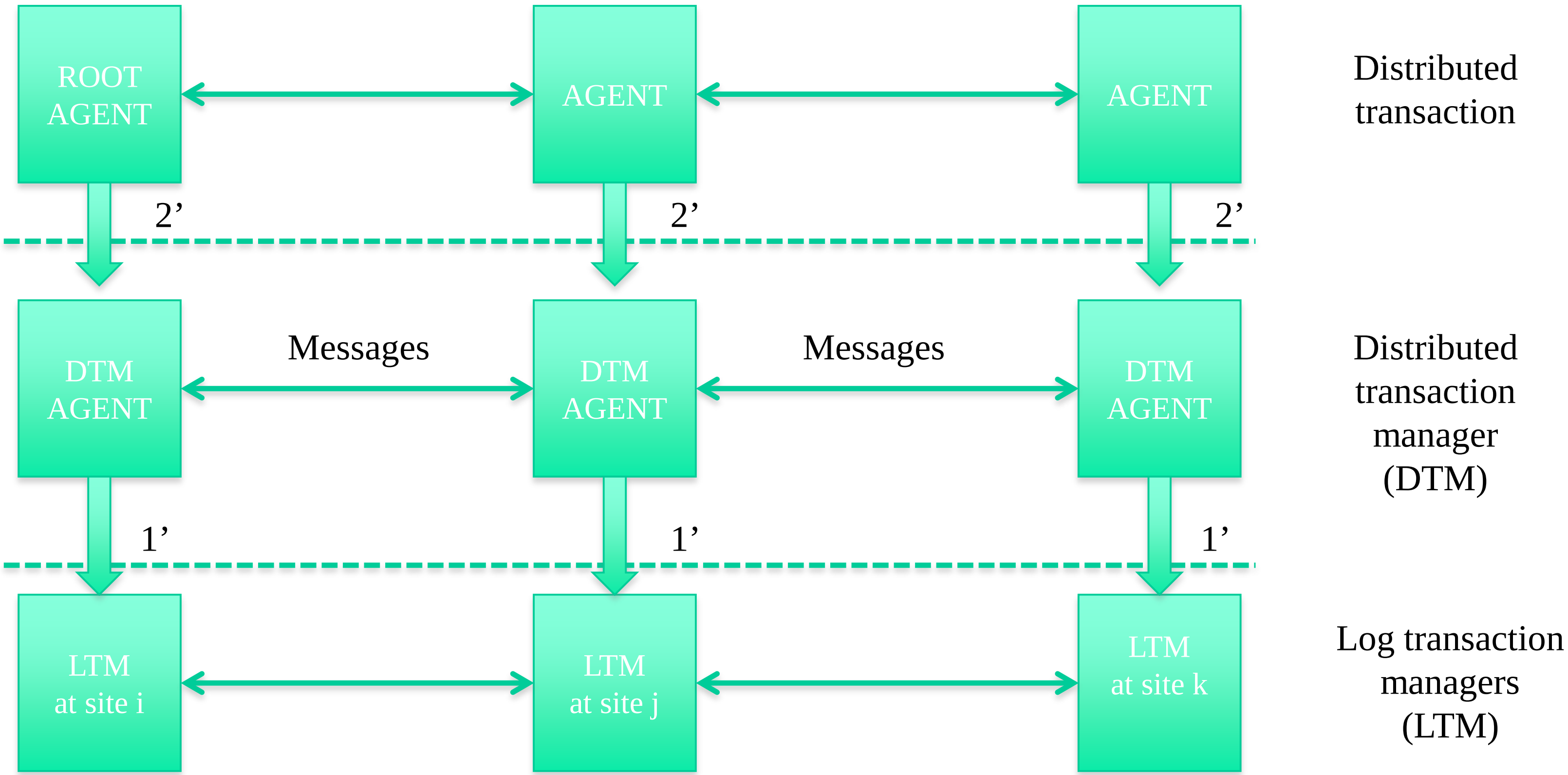
- In a general way, it can be **showed** that **resilient non-blocking protocols** to multiple **network partitions** do **not exist**.

4. Concurrency control for distributed transactions

Concurrency control for distributed transactions

- In this **section**:
  - ✓ We present **2-phase-locking**
    - The most widely used **technique** for **concurrency control**
  - ✓ We extend our **reference model** for the description of **concurrency control**
  - ✓ Theory of serializability
  - ✓ Proof of the correctness of 2-phase-locking
  - ✓ Deadlock detection and prevention
  - ✓ Timestamp based concurrency control algorithms

Reference model for distributed concurrency control



Interface 1’: Local_lock_shared, Local_lock_exclusive, Local_unlock
Interface 2’: Lock_shared, Lock_exclusive, Unlock

Concurrency control

- The **2 phase commitment protocol** provides for **atomicity** regarding **failures** and **durability**.
  - ✓ That's not enough!
  - ✓ **Concurrency control** is **needed** in order to proportionate **serializability** and **isolation**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Serializability in a centralized database

- A **transaction** accesses a **database** by issuing **read** and **write primitives**.

- Notation:
  - ✓ ***R(x,a)*** and ***W(x,a)*** denote a read and a write operation issued by a transaction ***T(x)*** on data item ***a*** .

- Example:
  - Transaction ***T(x)*** is ***a = b + c;***
  - ***T(x): R(x,b) < R(x,c) < W(x,a)***
  - < is the precedes **relation**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Schedule

- **Sequence of operations** performed by **transactions**

- **Example**
  - ✓ Consider the **transactions:**
    - ***T(x): a = b + c***;
    - ***T(y): d = b + c;***
  - ✓ **S1:** $R(x,b) < R(y,b) < R(x,c) < R(y,c) < W(x,a) < W(y,d)$
    - **S1** is a **concurrent schedule** since the operations of ***T(x)*** and ***T(y)*** are intercalated.
  - ✓ ***T(x)*** and ***T(y)*** are concurrently active

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Serial schedule

- A **schedule** is **serial** if **no transactions** execute **concurrently** in it

- **Example**
  - Consider the **transactions:**
    - ***T(x): a = b + c***;
    - ***T(y): d = b + c;***
  - **S2:** $R(x,b)<R(x,c)<W(x,a)<R(y,b)<R(y,c)<W(y,d)$

- **S2** can be written as a sequence of transactions of a serial schedule
    - **Serial(S2)**: $T(x) < T(y)$

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Serial schedule

- The **serial execution** of **transactions** which is described by a **serial schedule** is by definition **correct**.

- A **concurrent schedule** is **correct** if it is **serializable**
  - ✓ Computationally equivalent to a **serial schedule**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Conditions for schedules equivalence

- The following **two conditions** are sufficient to ensure that two **schedules** are **equivalent**:

Condition 1: Each **read operation** reads **data item** values which are produced by the same write operations in both **schedules**.

Condition 2: The final write operation on each data item is the same in both schedules.

Exercise

- Consider the **transactions:**
  - ✓ $\boldsymbol{T(x): a = b + c}$;
  - ✓ $\boldsymbol{T(y): d = b + c;}$
- Let $\boldsymbol{b = 2}$ and $\boldsymbol{c = 3}$.
  - ✓ Are **schedules S1** and **S2 equivalents**?

S1: $R(x,b=2) < R(y,b=2) < R(x,c=3) < R(y,c=3) < W(x,a=5) < W(y,d=5)$

S2: $\mathrm{R(x,b=2) < R(x,c=3) < W(x,a=5) < R(y,b=2) < R(y,c=3) < W(y,d=5)}$

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Conflicts

- **Two operations** are in **conflict** if they operate on the **same data** item
  - ✓ One of them is a **write operation**
    - The other one can be both **read** or **write**
  - ✓ They are issued by different **transactions**

Example of conflicts

- **Read-Write** conflict:
  - ***R(x,a), W(y,a)*** over item a
- **Write-Write** conflict:
  - ***W(x,a),W(y,a)*** over item a
- Transactions **x** and **y**:
  - *T(x): d = e + a;*
  - *T(y): a = b + c;*

S1: *R(x,e) < R(y,b) < R(**x,a**) < R(y,c) < W(x,d) < W(**y,a**)*

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Example of conflicts

- $\boldsymbol{S2}: R(y,b)<R(y,c)<W(y,a)<R(x,e)<R(x,a)<W(x,d)$
  - ✓ Notice that the **result** is **different** from **S1**

- Transactions **x** and **y**:
  - ✓ $T(x): a:=b+c;$
  - ✓ $T(y): a:=c+d;$

$$\boldsymbol{S1}: R(x,b)<R(y,c)<R(x,c)<R(y,d)<W(\mathbf{x,a})<W(\mathbf{y,a})$$

$$\boldsymbol{S2}: R(x,b)<R(y,c)<R(x,c)<R(y,d)<W(\mathbf{y,a})<W(\mathbf{x,a})$$

- Notice that the **result** is **different** from **S1**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Conflicts

- By using the notion of **conflict**, it is possible to state teh **sufficient condition** for the **equivalence** of **schedules** in a different way:
  - ✓ Two schedules ***S1*** and ***S2*** are **equivalent** if for each pair of **conflicting operations** O_i and O_j, such that O_i precedes O_j in ***S1*** then also O_i precedes O_j in ***S2***

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Locks

- **Lock** is a **concurrency control mechanism** used to ensure **serializability**.

- Lock modes
  - ✓ **Shared** mode
  - ✓ **Exclusive** mode

- A transaction is **well-formed**
  - ✓ If it always **locks** a data item in **shared mode** before reading it, **and**
  - ✓ It always locks a **data item** in **exclusive mode** before **writing** it

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Locks

- If one transaction obtain a **lock** in **exclusive mode** on a **data item:**
  - ✓ Other **transaction** cannot obtains the lock (**exclusive** or **shared**) on the **data item**.
- If one transaction obtains a **lock** on a **shared mode** on a **data item**
  - ✓ Other **transaction** cannot obtain the lock in **exclusive mode** on the data item.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Conflicts

- Two **transactions** are in **conflict** if they want to **lock** the **same data item** with two **incompatibles modes**
  - ✓ **Shared-exclusive** (Read-Write)
  - ✓ **Exclusive-Exclusive** (Write-Write)

- Locks solve conflicts (requests to hold an item with incompatible modes) by causing the transactions to wait. A **cyclic wait** leads to a deadlock

Conflicts

- A transaction, besides being **well-formed** must not request **new locks** after it has **released** one.
  - To ensure **serializability**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Scheme for 2-phase-lock (2PL)

- **Growing phase**:
  - ✓ Transaction **acquires** new **locks**.
- **Shrinking phase**:
  - ✓ Transaction **releases locks**.
- In order to guarantee **isolation** we must therefore require that **transactions** hold all their **exclusive locks** until **commitment**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Correctness proof of 2PL

- When a **schedule** is **not serializable** in terms of lock?

- Transaction ***T(x)*** and ***T(y)***
  - ✓ *T(x): a = b + c;*
  - ✓ *T(y): b = a + d;*

S1: R(**x,b**)<R(**y,a**)<R(x,c)<R(y,d)<W(**x,a**)<W(**y,b**)

- ***T(y)*** waits ***T(x)*** on **b** and ***T(x)*** waits ***T(y)*** on **a**

Correctness proof of 2PL

- In a general way, a **schedule** of ***n*** **transactions** is **serializable** if there are a **set** o ***n*** pairs of **conflicting operations** such as:

$$O(1,a) < O(2,a)$$

$$O(2,b) < O(3,b)$$

$$\ldots$$

$$O(n\text{-}1,i) < O(n,i)$$

$$O(n,j) < O(1,j)$$

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Correctness proof of 2PL

- Suppose that **each transaction** has **acquired the lock** on the "***left side***"
  - ✓ Now, each transaction will indefinitely **wait** to acquire the lock on the "***right side***"
    - Since in **2PL** scheme, **locks** are note **released** during the growing phase.
  - ✓ Consequently, the **system** enters in **deadlock** and the **non-serializable schedule** cannot occur.

Correctness proof of 2PL

- **Note**:
  - ✓ **Deadlocks** can occur and consequently a **transaction** must employ some method for **detection** and deadlock resolution.

Correctness proof of 2PL

- Which of the following **schedules** are **correct (**i.e. **serializable)** and, for the **serializable schedule**, what would be a equivalent serial **schedule**?

- Transactions ***T(x)*** and ***T(y)***
  - ✓ *T(x): a = b + c;*
  - ✓ *T(y): b = a + c;*

S3: R(x,b) < R(y,a) < R(x,c) < R(y,c) < W(y,b) < W(x,a)

S4: R(y,c) < R(x,c) < R(y,a) < W(y,b) < R(x,b) < W(x,a)

Correctness proof of 2PL

- The **equivalent serial schedule** can be conceived as giving the **total ordering** of the **transactions**

Serializability in a distributed database

- **Each transaction** performs **operations** at several **sites**.

- An execution of ***n*** **distributed transactions** ***T(1)..T(n)*** at **m sites** is modeled by a set of **local schedules** ***S1..Sm***

- Generally, each **site manages** specific **data items**.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Serializability in a distributed database

- Let ***T(1)*** be composed of **sub-transactions**:
  - ✓ $a = a + 1$ at **site 1**
  - ✓ **b = b + 1** at **site 2**

- And let ***T(2)*** be composed of **sub-transactions**:
  - ✓ $a = a / 2$ at **site 1**
  - ✓ $b = b / 2$ at **site 2**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Serializability in a distributed database

- **Site 1** manages a while **site 2** manages **b**.
- **Verify** the **values** of ***a*** and ***b*** after ***T(1) < T(2)*** and ***T(2) < T(1)***.
- Suppose that *a = b = 0*
  - ✓ *T(1) < T(2) =>* ***a = b =1/2***
  - ✓ *T(2) < T(1) =>* ***a = b = 1***

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Serializability in a distributed database

- Suppose the following **execution** of ***T(1)*** and ***T(2)*** at two **sites**

S1: R(1,a) < W(1,a) < R(2,a) < W(2,a)

S2: R(2,b) < W(2,b) < R(1,b) < W(1,b)

Serializability in a distributed database

- Each **local schedule** is **serializable**, i.e.
  - ✓ ***S1:*** *T(1) < T(2)*
  - ✓ ***S2:*** *T(2) < T(1)*
- However, there is no a **global** and **total ordering** of **transactions**
- If the execution happened, ***a=1/2*** and ***b=1***.
  - ✓ **Incorrect** result!
- Remember that in order a **distributed execution of transactions** to be **correct**, it must be **serializable**

Correctness condition for distributed transactions

- Let ***E*** one execution of transactions ***T(1)..T(n)*** modelled by **schedules *S1..Sm***.
  - **E** is **correct** (serializable) if:
    - there is a **total ordering** such that for **each pair** of **conflicting operations *O(i)*** and ***O(j)*** from ***T(i)*** and ***T(j)***, ***O(i) < O(j) in any schedule S1..Sm*** if and only if ***T(i)* < *T(j)*** in the **total ordering**.

Correctness condition for distributed transactions

- Without no surprise, the **2PL scheme** is a **correct concurrency control mechanism** for **distributed transactions.**
  - ✓ The **proof** is as before.

Which of the following executions are serializable?

- **Execution 1:**
  - ✓ S1: R(1,a) < R(2,a) < W(2,b) < W(1,a)
  - ✓ S2: R(1,c) < R(2,d) < W(2,c) < W(1,c)
- **Execution 2:**
  - ✓ S1: R(1,a) < R(2,a) < W(2,b) < W(1,b)
  - ✓ S2: W(1,d)
- **Execution 3:**
  - ✓ S1: R(1,a) < R(2,a) < W(1,a) < W(2,b)
  - ✓ S2: R(1,d) < R(2,d) < W(2,d) < W(1,c)
- **Execution 4:**
  - ✓ S1: R(1,b) < R(2,a) < W(2,a)
  - ✓ S2: W(2,d) < R(1,c) < R(2,c) < W(1,c)

Comments about serializability

- **Given**:
  - ***T(1):*** transfer \$10 from a to b (a,b initially 100)
  - ***T(2):*** transfer \$20 from b to a

- The **non-serializable execution** produces the **correct** result (i.e. ***a = 110***, ***b = 90***)
  - ***S1:*** *R(1,a) < W(1,a) < R(2,a) < W(2,a)*
  - ***S2:*** *R(2,b) < W(2,b) < R(1,b) < W(1,b)*

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Comments about serializability

- **Serializability** is the **weakest criteria** in order to preserve **consistency** if **no semantic information** about the **transaction** is available.

- It could be **defined** a higher level of different **read** and **write** operations to capture more **semantic knowledge**.

- For instance
  - ✓ Increment **I(x,d):** $x = x + d;$
  - ✓ Decrement **D(x,d):** $x = x - d;$

Comments about serializability

- These operations do not conflict since:
  - ✓ ***I(x,u) < D(x,w)*** is equivalent to ***D(x,w) < I(x,u)***
  - ✓ The result is the same:
    - ***x-w+u***

- We can show the transaction as:
  - ✓ ***S1:*** *D(1, a, 10) < I(2, a, 20)*
  - ✓ ***S2:*** *D(2, b, 20) < I(1, b, 10)*

- This **approach** has been used in some **operating systems** to **achieve** more **concurrency**.

Concurrency control based on timestamp

- The **2PL scheme** has the **disadvantage** of holding the **records using locks** until the **transaction** completion.
  - ✓ If the **transaction** has a **long duration**, the delay can result in a **large number** of **locks**
    - Hence, increasing the chance of **deadlock**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Concurrency control based on timestamp

- The **concurrency control** based on **timestamp** does not use **locks**.

- In order to proportionate a **total order** in the **transactions**, a **unique timestamp** is assigned to each **transaction**.
  - ✓ The **Lamport's algorithm** for **logical clocks** is used for the generation of **timestamps**.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Basic mechanism of timestamp

1. Each **transaction** is assigned a **unique timestamp** at its site of origin.

2. Each **read** and **write operation** has the **timestamp** ***TS*** of its **transaction**.

3. Each data item (x) has the following information:
   - ✓ ***WTM(x)*** – the biggest timestamp of a write operation on ***x***.
   - ✓ ***RTM(x)*** - the biggest timestamp of a **read operation** on ***x***.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Basic mechanism of timestamp

4. For **read** operations:

If (***TS* < *WTM(x)***) then

Reject the **read operation** and restart the transaction

else

Execute **read** and ***RTM(x):= max(RTM(x), TS);***

Basic mechanism of timestamp

5. For **write operations**:

If (***TS < RTM(x)***) or (***TS < WTM(x)***) then

  Reject the **write operation** and **restart** the **transaction**

else

  execute **write** and ***WTM(x):= TS***;

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Basic mechanism of timestamp

- The **steps 4** and **5** ensure that **conflicting operations** are executed in a **timestamped order** at all **sites**
  - ✓ Hence, the **correctness condition** for **serializability** is satisfied.

Basic mechanism of timestamp

- **Notes**:
  - A **restarted transaction** will acquire a **bigger timestamp** and therefore will eventually continue sucessfully.
  - **Transactions** are never **blocked** (they are restarted) amd therefore, **deadlock** can not occur.
  - The **disadvantage** of this scheme is the **cost** of **restarting** the **transactions**

Timestamp concurrency control and 2-Phases commitment scheme

- The **2-phases commitment scheme** requires the existence of a **time interval** on which the agents can **commit** or **abort transactions**.
  - ✓ With the condition that all **exclusive locks** are safe until the **transaction commitment**.
  - ✓ It guarantess **isolation**
    - No visibility of **intermediate results**

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
→ 4. Concurrency control for distributed transactions

Timestamp concurrency control and 2-Phases commitment scheme

- With **timestamps**, **prewrites** are used instead of **exclusive locks**.
  - Instead of doing a **write operations**, transactions do **prewrite operations** to **buffer** and not to **update**
  - Only when the **transactions commit**, their correspondent **updates** are **applied** to the **database**.

1. Introduction
2. A framework for transaction management
3. Supporting atomicity of distributed transactions
4. Concurrency control for distributed transactions

Timestamp concurrency control and 2-Phases commitment scheme

- The **timestamp mechanism** must be lightly modified taking into account **pending prewrites**.

References

- Ceri, Stefano, and Giuseppe Pelagatti. **Distributed databases principles and systems**. McGraw-Hill, Inc., 1984.
  - ✓ Chapter 7: The management of distributed transactions
  - ✓ Chapter 8: Concurrency control
  - ✓ Chapter 9: Reliability