Distributed Systems Part II:
Distributed Transactional Systems

Dr.-Ing. Thomas Hornung

Lehrstuhl für Datenbanken und Informationssysteme
Universität Freiburg

SS 2013

Outline

(1) Introduction

(2) Distributed Systems Architectures

(3) Transaction Model

(4) Distributed Concurrency Control

(5) Reliability

(6) Modeling and Analysis of Distributed Applications

(7) Replication

(8) Distributed Processing of “Big Data”

Organization

- Time and location:
  - Course: Monday, 14:00 - 16:00 c.t., Room 101-01-009/013
    Friday, 14:00 - 16:00 c.t., Room 101-01-009/013, every two weeks
  - Exercises: Friday, 14:00 - 16:00 c.t., Room 101-01-009/013, every two weeks
- The final examination is oral. Please register on-line using the campus management system.
- There are no requirements for the registration and please observe the registration deadline.

All material about lectures and exercises can be found at the web page for Part 1 of the course!

Literature:

- G. Weikum, G. Vossen. *Transactional Information Systems*. Morgan Kaufmann, 2002.
- P. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987. Download: http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx
- M.T. Özsu, P. Valduriez. *Principles of Distributed Database Systems*. Springer Verlag, 2001, 3Ed.
- T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2012, 3Ed.

Chapter 1: Introduction

Why *transactional?*

Transactions form a reasonable *abstraction concept* for many classes of real-life data processing problems.

- Transactions cope in a very elegant way with the subtle and often difficult issues of keeping data consistent in the presence of highly concurrent data accesses and despite all sorts of failures.
- This is achieved in a generic way invisible to the application logic so that application developers are freed from the burden of dealing with such system issues.
  - The application program simply has to indicate the boundaries of a transaction by issuing `BEGIN TRANSACTION` and `COMMIT TRANSACTION` calls.
  - The execution environment considers all requests receiving from the application programs execution within this dynamic scope as belonging to the same transaction.
  - For the transaction's requests and effects on the underlying data certain properties are guaranteed: ACID properties.

Challenges inherent to the transaction concept demonstrated by some examples

(Expl.1a) Debit/credit

Consider a debit/credit-program of a bank which transfers a certain amount of money between two accounts. Executing the program will give us the following transaction T:

```
BEGIN
% Withdraw
READ current value VA of account A from disk into T's local main memory;
decrement VA by amount X;
WRITE new value VA' = VA - X of account A from T's local main memory onto disk;
% Deposit
READ current value VB of account B from disk into T's local main memory;
increment VB by amount X;
WRITE new value VB' = VB + X of account B from T's local main memory onto disk;
COMMIT;
```

- Assume when executing T the system runs into a failure, e.g. after writing A and before reading B. A customer of the bank has lost X money!
- Assume debit/credit-transaction T1 is running concurrently to a transaction T2, which computes the balance of the accounts A and B. Then the READ and WRITE accesses of both transactions may be interleaved. Assume that T2 is executed after T1 writing A and before T1 writing B, then the balance computed will be incorrect.

(Expl.1b) Distributed debit/credit

Assume that different branches of the bank are involved, where each branch maintains its own server. Assume further, at Branch1 a debit/credit-transaction is started and at Branch2 a balancing transaction, where both involve the same accounts. Transactions shall have access to accounts on remote server via remote procedure calls (RPC), a synchronous communication mechanism transparent to the programmer. We assume procedures *withdraw*(*account*, *amount*), *deposit*(*account*, *amount*) and *getBalance*(*account*).

A possible interleaving when both transactions are running in parallel.

Branch1(*accountA*)	*Branch2*(*accountB*)
T_1 : withdraw(A,10)	
T_1 : call(deposit(B,10))	
	T_2 : getBalance(B)
	T_2 : call(getBalance(A))
	T_1 : deposit(B,10)
T_2 : getBalance(A)	
	T_2 : display A+B

time

A
50
40

B
50
50

40

60

90

→ 100

An incorrect balance will be displayed!

(Expl.1c) Distributed debit/credit

Assume that different branches of the bank are involved, where each branch maintains it own servers. Assume further, at Branch1 a debit/credit-transaction is started and at Branch2 a balancing transaction is started, where both involve the same accounts. Finally assume, that each transaction implements exclusive access to both accounts during execution. Communication is explicitly implemented by exchanging messages between the involved servers.

A possible interleaving when both transactions are running in parallel.

*Branch*1	*Branch*2	
T_1 : {{lock(A); withdraw(A,10)} || {send {lock(B); deposit(B,10)} to Branch2}}		
	T_2 : {{lock(B); getBalance(B)} || {send {lock(A); getBalance(A)} to Branch1}}	*time*
T_1 : {wait for ACK of deposit at Branch2} T_2 : {wait until lock(A) granted}		
	T_2 : {wait for balance of A} T_1 : {wait until lock(B) granted}	

A deadlock has occured which is difficult to detect!

(Expl.2) Electronic commerce

Consider the following purchasing activity, which covers several different servers located at different sites:

- A client connects to a bookstore's server and starts browsing and querying the catalog.
- The client gradually fills a shopping card with items intended to purchase.
- When the client is about to check out she makes final decisions what to purchase.
- The client provides all necessary information for placing a legally binding order, e.g. shipping address and credit card.
- The merchants's server forwards the payment information to the customer's bank or to a clearinghouse. When the payment is accepted, the inventory is updated, shipping is initiated and the client is notified about successful completion of her order.

- The final step of the purchasing is the most critical one. Several servers maintained by different institutions are involved.
- Most importantly it has to be guaranteed, that either all the tasks of the final step are processed correctly, or the whole purchasing activity is undone.

(Expl.3) Mobile computing

Assume that the described purchasing activity is performed via a smartphone. Then the described picture is even more complicated.

- The smartphone might be temporarily disconnected from the mobile net. Thus it is not guaranteed, that the state of the catalog as seen by the client reflects the state of the catalog at the server.
- If the client enters a dead spot during processing of the final step of the purchasing activity, confusion may arise, e.g. the purchasing is started again resulting in double orders.

Transaction Concept

ACID properties

- A tomicity: A transaction is executed completely or not at all.
- C onsistency: Consistency constraints defined on the data are preserved.
- I solation: Each transaction behaves as if it were operating alone on the data.
- D urability: All effects will survive all software and hardware failures.

$\Longrightarrow$ *Concurrency Control* (I) and *Recovery* (A, D) provide the mechanisms needed to cope with the problems demonstrated by Expl.1-3.

Concurrency Control Refresh

Basics

- Set of transactions $\mathcal{T} = \{T_1, \dots, T_n\}$.
- A transaction is given as a sequence of read (R) - and write (W)-actions over database objects $\{A, B, C, \dots\}$, e.g.

$$\begin{aligned} T_1 &= R_1A\ W_1A\ R_1B\ W_1B \\ T_2 &= R_2A\ W_2A\ R_2B\ W_2B \\ T_3 &= R_3A\ W_3B \end{aligned}$$

- Let WX be the j-th action of transaction T and assume that $RA_1, \dots, RA_n$ are the read actions of T being processed in the indicated order before WX. Then the value of X written by T is given by $f_{T,j}(a_1, \dots, a_n)$, where $f_{T,j}$ is an arbitrary, however unknown function and the a's are the values read in the indicated order by the preceding read actions.
- A concurrent execution of a set of transactions is called schedule and is given as a - possibly interleaved - sequence of the respective actions, e.g.

$$\begin{aligned} S_1 &= R_1A\ W_1A\ R_3A\ R_1B\ W_1B\ R_2A\ W_2A\ W_3B\ R_2B\ W_2B \\ S_2 &= R_1A\ W_1A\ R_3A\ R_1B\ W_1B\ R_2A\ W_2A\ W_3B\ R_2B\ W_2B \\ S_3 &= R_3A\ R_1A\ W_1A\ R_1B\ W_1B\ R_2A\ W_2A\ R_2B\ W_2B\ W_3B \end{aligned}$$

- A schedule is called serial, if it is not interleaved.

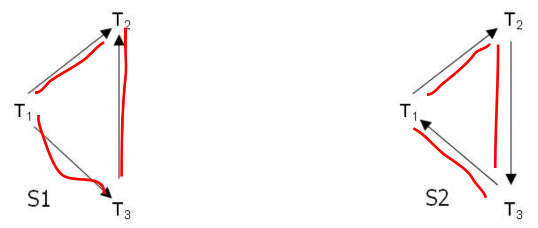
Correctness

- A schedule is called (conflict-)serializable,[1] if there exists a (conflict-)equivalent serial schedule over the same set of transactions.
- For a given schedule S over a set of transactions, the conflict graph $G(S)$ is given as $G(S) = (V, E)$, where the node set V is the set of transactions in S and the set of edges E is given by so called conflicts as follows:
  - $S = \ldots W_iA \ldots R_jA \ldots \;\Rightarrow\; T_i \rightarrow T_j \in E$, if there is no other write-action to A between W_iA und R_jA in S.
  - $S = \ldots W_iA \ldots W_jA \ldots \;\Rightarrow\; T_i \rightarrow T_j \in E$, if there is no other write-action to A between W_iA und W_jA in S.
  - $S = \ldots R_iA \ldots W_jA \ldots \;\Rightarrow\; T_i \rightarrow T_j \in E$, if there is no other write-action to A between R_iA und W_jA in S.
- A schedule is serializable, iff its conflict graph is acyclic.

[1] We consider only conflict-serializability and therefore talk about serializability in the sequel, for short.

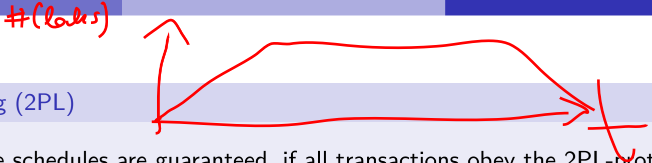
Example

Schedule S_1: R_1A W_1A R_3A R_1B W_1B R_2A W_2A W_3B R_2B W_2B
Schedule S_2: R_3A R_1A W_1A R_1B W_1B R_2A W_2A R_2B W_2B W_3B



S_1 is serializable, S_2 is not.

To exclude not serializable schedules, a so called *transaction manager* enforces certain transaction behaviour.



2-Phase Locking (2PL)

- Serializable schedules are guaranteed, if all transactions obey the 2PL-protocol:
  - For each transaction T, each RA and WA has to be surrounded by a lock/unlock pair LA, UA:

$$T = \ldots \boxed{R/WA} \ldots \Longrightarrow T = \ldots \boxed{LA} \ldots R/WA \ldots \boxed{UA \ldots}$$

  - For each A read or written in T there exists at most one pair LA and UA.
  - For each T and any LA_1, UA_2 there holds: $T = \ldots LA_1 \ldots UA_2 \ldots$.
    $\Longrightarrow$ No more locking after the first unlock!
  - In any schedule S, the same object A cannot be locked at the same time by more than one transaction:

$$S = \ldots \underline{L_iA \ldots L_jA} \ldots \Longrightarrow S = \ldots L_iA \ldots \boxed{U_iA} \ldots L_jA \ldots$$

- Every schedule according to 2PL is serializable, however
  - Not every serializable schedule can be produced by 2PL.
  - Deadlocks may occur.

Example 1

$$T_1 = L_1A\ R_1A\ L_1B\ U_1A\ W_1B\ U_1B,$$
$$T_2 = L_2A\ R_2A\ W_2A\ U_2A,$$
$$T_3 = L_3C\ R_3C\ U_3C.$$

$$S = L_1A\ R_1A\ L_1B\ U_1A\ L_2A\ R_2A\ L_3C\ R_3C\ U_3C\ W_1B\ U_1B\ W_2A\ U_2A$$

Example 2

$$T_1 = L_1A\ R_1A\ L_1B\ U_1A\ W_1B\ U_1B,$$
$$T_2 = L_2A\ R_2A\ W_2A\ U_2A,$$
$$T_3 = L_3C\ R_3C\ U_3C.$$

$$S = L_1A\ R_1A\ L_1B\ U_1A\ L_2A\ R_2A\ L_3C\ R_3C\ U_3C\ W_1B\ U_1B\ W_2A\ U_2A$$

The *lock point* of a transaction using 2PL is given by the first unlock of the transaction.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \dots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \to T_2 \to \dots \to T_k \to T_1$.

- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T preceds the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \dots, A_k$ are data items:

$$\begin{aligned} S &= \dots U_1 A_1 \dots L_2 A_1 \dots, \\ &\vdots \\ S &= \dots U_{k-1} A_{k-1} \dots L_k A_{k-1} \dots, \\ S &= \dots U_k A_k \dots L_1 A_k \dots. \end{aligned}$$

- Let $l_1, \dots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\dots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k \rightarrow T_1$.

- Each edge $T \rightarrow T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \ldots, A_k$ are data items:

$$\begin{aligned} S &= \ldots U_1 A_1 \ldots L_2 A_1 \ldots, \\ &\vdots \\ S &= \ldots U_{k-1} A_{k-1} \ldots L_k A_{k-1} \ldots, \\ S &= \ldots U_k A_k \ldots L_1 A_k \ldots. \end{aligned}$$

- Let $l_1, \ldots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\ldots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $\underline{T_1} \rightarrow \underline{T_2} \rightarrow \cdots \rightarrow \underline{T_k} \rightarrow \underline{T_1}$.

- Each edge $T \rightarrow T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \ldots, A_k$ are data items:

$$
\begin{aligned}
S &= \ldots \underline{U_1 A_1} \ldots \underline{L_2 A_1} \ldots, \\
&\vdots \\
S &= \ldots \underline{U_{k-1} A_{k-1}} \ldots \underline{L_k A_{k-1}} \ldots, \\
S &= \ldots \boxed{U_k A_k} \ldots \underline{L_1 A_k} \ldots .
\end{aligned}
$$

- Let $l_1, \ldots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\ldots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \to T_2 \to \cdots \to T_k \to T_1$.

- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \ldots, A_k$ are data items:

$$\begin{array}{l} S = \ldots U_1 A_1 \ldots L_2 A_1 \ldots, \\ \vdots \\ S = \ldots U_{k-1} A_{k-1} \ldots L_k A_{k-1} \ldots, \\ S = \ldots U_k A_k \ldots L_1 A_k \ldots. \end{array}$$

- Let $l_1, \ldots, l_k$ be the lock points of the involved transactions. Then we have l_1 before l_2, $\ldots$, l_{k-1} before l_k and l_k before l_1. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

2PL guarantees serializability of schedules.

Let S be a schedule of a set of 2PL-transactions $\mathcal{T} = \{T_1, \dots, T_n\}$.

Assume, S is not serializable, i.e. the conflict graph $G(S)$ is cyclic, where w.l.o.g. $T_1 \to T_2 \to \dots \to T_k \to T_1$.

- Each edge $T \to T'$ implies T and T' having conflicting actions, where the action of T precedes the one of T'.
- Because of surrounding actions by lock/unlock and the 2PL-rule, T' can execute its action only after the lock-point of T. This implies the following structure of S, where $A_1, \dots, A_k$ are data items:

$$
\begin{aligned}
S &= \dots \underline{U_1 A_1} \dots L_2 A_1 \dots, \\
&\vdots \\
S &= \dots \underline{U_{k-1} A_{k-1} \dots L_k} A_{k-1} \dots, \\
S &= \dots \underline{U_k A_k} \dots L_1 A_k \dots .
\end{aligned}
$$

- Let $\underline{l_1, \dots, l_k}$ be the lock points of the involved transactions. Then we have $\underline{l_1}$ before $\underline{l_2}$, $\dots$, l_{k-1} before l_k and $\underline{l_k \text{ before } l_1}$. However this is a contradiction to the structure of a schedule. Therefore S is serializable.

Recovery Refresh

Basics

- Reliability has to be achieved even though system components are unreliable, in general.
- Log files are a prerequisite for recovery from system failures.
- Log files are maintained as follows:
    - When a transaction T starts executing, $(T, Begin)$ is written into the log.
    - For each *WA* of a transaction T, (T, A, A_{old}, A_{new}) is written into the log, where A_{new} is the new value (After-image) and A_{old} the previous value (before-image) of A.
    - When a transaction commits its execution, $(T, Commit)$ is written into the log and otherwise $(T, Abort)$.
- The write-ahead-log-rule (WAL) has to be observed: writing into the log must preceed writing into the database.

Consequence of atomicity

- Whenever a transaction has processed a commit action, all its effects are permanent and will survive all failures.
- Whenever a transaction has processed a abort action - respectively is aborted -, all its effects are removed.

Recovery from system failures: Backwards Restart-Algorithm, logging has to be done on page-level

- $Redone := \emptyset$; $Undone := \emptyset$.
- The log is processed backwards. Let (T, A, A_{old}, A_{new}) the next log-entry to be considered. If $A \notin Redone \cup Undone$:

  Redo: If $(T, Commit)$ has already been found, then process WA with value A_{new} and perform $Redone := Redone \cup \{A\}$.

  Undo: Otherwise perform WA with value A_{old} and perform $Undone := Undone \cup \{A\}$.

Example

System-failure | State after Restart

T_1 LA RA WA CO UA

T_2 LB RB LA RA WB CO UA,B

T_3 LC RC WC

WA not yet materialized in the database, e.g. read accesses RA are expected

DB :
A_0
B_0
C_0

$f_2(f_1(A_0),B_0)$ $f_3(C_0)$

State after Restart:
$f_1(A_0)$
$f_2(f_1(A_0),B_0)$
C_0

localmemory/
systembuffer :
T_1 $f_1(A_0)$
T_2 $f_2(f_1(A_0),B_0)$
T_3 $f_3(C_0)$

Log (reduced):
$(T_1, A, A_0, f_1(A_0)), (T_1, CO), (T_2, B, B_0, f_2(f_1(A_0), B_0)), (T_2, CO), (T_3, C, C_0, f_3(C_0))$