PAYAP UNIVERSITY

Transactions and Concurrency control

Vaidė Narváez

Computer Information Systems

September 6th, 2010

PAYAP UNIVERSITY

Introduction

- The goal of transactions: the objects managed by a server must remain in a consistent state
  - when they are accessed by multiple transactions and
  - in the presence of server crashes
- Recoverable objects
  - can be recovered after their server crashes
  - objects are stored in permanent storage
- Failure model: transactions deal with crash failures of processes and omission failures of communication
- Designed for an asynchronous system: It is assumed that messages may be delayed

PAYAP UNIVERSITY

Running Example

Account

deposit(amount)

deposit amount in the account

withdraw(amount)

withdraw amount from the account

getBalance() → amount

return the balance of the account

setBalance(amount)

set the balance of the account to amount

Branch

create(name) → account

create a new account with a given name

lookUp(name) → account

return a reference to the account with the given name

branchTotal() → amount

return the total of all the balances at the branch

PAYAP UNIVERSITY

Atomic operations at the server

(no transactions)

- when a server uses multiple threads it can perform several client operations concurrently
- if we allowed deposit and withdraw to run concurrently we could get inconsistent results
- objects should be designed for safe concurrent access e.g. in Java use synchronized methods, e.g.

```
public synchronized void deposit(int amount) throws
RemoteException
```

- atomic operations are free from interference from concurrent operations in other threads.
- use any available mutual exclusion mechanism (e.g. mutex)

PAYAP UNIVERSITY

Client cooperation

- Clients share resources via a server, e.g., some clients update server objects and others access them
- servers with multiple threads require atomic objects, but in some applications, clients depend on one another to progress, e.g. consumer-producer paradigm
- it would not be a good idea for a waiting client to poll the server to see whether a resource is yet available, it would also be unfair (later clients might get earlier turns) Why not?
- Java `wait` and `notify` methods allow threads to communicate with one another and to solve these problems, e.g. when a client requests a resource, the server thread waits until it is notified that the resource is available

PAYAP UNIVERSITY

Failure model

- Algorithms work correctly when predictable faults occur, but if a disaster occurs, we cannot say what will happen
- Writes to permanent storage may fail
  - e.g. by writing nothing or a wrong value (write to wrong block is a disaster)
  - reads can detect bad blocks by checksum
- Servers may crash occasionally.
  - when a crashed server is replaced by a new process its memory is cleared and then it carries out a recovery procedure to get its objects state
  - faulty servers are made to crash so that they do not produce arbitrary failures
- There may be an arbitrary delay before a message arrives. A message may be lost, duplicated or corrupted.
  - recipient can detect corrupt messages (by checksum)
  - forged messages and undetected corrupt messages are disasters

PAYAP UNIVERSITY

Transactions

- Some applications require a sequence of client requests to a server to be atomic in the sense that:
  1. they are free from interference by operations being performed on behalf of other concurrent clients; and
  2. either all of the operations must be completed successfully or they must have no effect at all in the presence of server crashes.

- Transactions originate from database management systems

- Transactions apply to recoverable objects and are intended to be atomic.

PAYAP UNIVERSITY

Banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

- This transaction specifies a sequence of related operations involving bank accounts named *A*, *B* and *C* and referred to as *a*, *b* and *c* in the program
- the first two operations transfer $100 from *A* to *B*
- the second two operations transfer $200 from *C* to *B*

PAYAP UNIVERSITY

Atomicity of transactions

- All or nothing:
  - it either completes successfully, and the effects of all of its operations are recorded in the objects, or (if it fails or is aborted) it has no effect at all. This all-or-nothing effect has two further aspects of its own:
    - *failure atomicity:* the effects are atomic even when the server crashes;
    - *durability:* after a transaction has completed successfully, all its effects are saved in permanent storage.

- Isolation:
  - Each transaction must be performed without interference from other transactions - there must be no observation by other transactions of a transaction's intermediate effects

Transaction life

Successful	*Aborted by client*	*Aborted by server*	
openTransaction *operation* *operation*	*openTransaction* *operation* *operation*		*openTransaction* *operation* *operation*
⋮	⋮	server aborts transaction →	⋮
operation	*operation*		*operation ERROR reported to client*
closeTransaction	*abortTransaction*		

Figure 12.4

- A transaction is either successful (it commits)
  - the coordinator sees that all objects are saved in permanent storage
- or it is aborted by the client or the server
  - make all temporary effects invisible to other transactions
  - how will the client know when the server has aborted its transaction?

PAYAP UNIVERSITY

Concurrency control

- **The lost update problem** occurs when two transactions both read the old value of a variable and use it to calculate a new value

- **Inconsistent retrievals** occur when a retrieval transaction observes values that are involved in an ongoing updating transaction

PAYAP UNIVERSITY

The lost update problem

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance();		*balance = b.getBalance();*	
*b.setBalance(balance*1.1);*		*b.setBalance(balance*1.1);*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

- the initial balances of accounts *A, B, C* are $100, $200, $300
- both transfer transactions increase *B*'s balance by 10%

PAYAP UNIVERSITY

Inconsistent retrievals

Transaction *V*: *a.withdraw(100)* *b.deposit(100)*		**Transaction *W*:** *aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

- *V* transfers $100 from *A* to *B* while *W* calculates branch total (which should be $600)

PAYAP UNIVERSITY

Serial Equivalence

- if each one of a set of transactions has the correct effect when done on its own, then if they are done one at a time in some order the effect will be correct

- a *serially equivalent* interleaving is one in which the combined effect is the same as if the transactions had been done one at a time in some order

- the same effect means:
  - the read operations return the same values
  - the instance variables of the objects have the same values at the end

PAYAP UNIVERSITY

Serial Equivalence

lost update cured

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

- if one of *T* and *U* runs before the other, they cant get a lost update,
- the same is true if they are run in a serially equivalent ordering

PAYAP UNIVERSITY

Serial Equivalence

inconsistent retrieval cured

Transaction *V*: *a.withdraw(100);* *b.deposit(100)*		**Transaction *W*:** *aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		total = total+c.getBalance()	
		...	

- if *W* is run before or after *V*, the problem will not occur
- therefore it will not occur in a serially equivalent ordering of *V* and *W*
- the illustration is serial, but it need not be

PAYAP UNIVERSITY

Conflicting operations

- **a pair of operations conflicts** if their combined effect depends on the order in which they were performed. e.g. read and write (whose effects are the result returned by read and the value set by write)

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

PAYAP UNIVERSITY

Serial equivalence

For two transactions to be serially equivalent, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access

- Consider:
  - $T : x = read(i); write(i, 10); write(j, 20);$
  - $U : y = read(j); write(j, 30); z = read(i);$
- Serial equivalence requires that either
  1. T accesses i before U and T accesses j before U. or
  2. U accesses i before T and U accesses j before T
- Serial equivalence is used as a criterion for designing concurrency control schemes

Non-serially equivalent interleaving

Transaction *T*:	Transaction *U*:
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

- Each transactions access to *i* and *j* is serialised w.r.t one another, but
- *T* makes all accesses to *i* before *U* does
- *U* makes all accesses to *j* before *T* does

not serially equivalent

PAYAP UNIVERSITY

Recoverability from aborts

If a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects

- dirty reads: an interaction between a read operation in one transaction and an earlier write operation on the same object (by a transaction that then aborts)
  a transaction that committed with a *dirty read* is not recoverable

- premature writes: interactions between write operations on the same object by different transactions, one of which aborts

PAYAP UNIVERSITY

A dirty read

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	$100		
a.setBalance(balance + 10)	$110		
		balance = a.getBalance()	$110
		a.setBalance(balance + 20)	$130
		commit transaction	
abort transaction			

PAYAP UNIVERSITY

Recoverability of transactions

- If a transaction (like *U*) commits after seeing the effects of a transaction that subsequently aborted, it is not *recoverable*

For recoverability:

A commit is delayed until after the commitment of any other transaction whose state has been observed

e.g. *U* waits until *T* commits or aborts.
if *T* aborts then *U* must also abort

PAYAP UNIVERSITY

Cascading aborts

- Cascading aborts:
  - Suppose that *U* delays committing until after *T* aborts.
  - then, *U* must abort as well.
  - if any other transactions have seen the effects due to *U*, they too must be aborted.
  - the aborting of these latter transactions may cause still further transactions to be aborted.

To avoid cascading aborts:

Transactions are only allowed to read objects written by committed transactions. To ensure this, any *read* operation must be delayed until other transactions that applied a *write* operation to the same object have committed or aborted.

Avoidance of cascading aborts is a stronger condition than recoverability

PAYAP UNIVERSITY

Premature writes

Transaction *T*:		Transaction *U*:
a.setBalance(105)	before *T* and *U* the balance of A was $100	*a.setBalance(110)*
	$100	
a.setBalance(105)	$105	
		a.setBalance(110) $110

serially equivalent executions of T and U

interaction between *write* operations when a transaction aborts

- some database systems keep "before images" and restore them after aborts.
  - e.g. $100 is before image of *T*'s write, $105 is before image of *U*'s write
  - if *U* aborts we get the correct balance of $105,
  - But if *U* commits and then *T* aborts, we get $100 instead of $110

PAYAP UNIVERSITY

Strict executions

- Curing premature writes:
  - if a recovery scheme uses before images, write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted
- Strict executions of transactions
  - to avoid both *dirty reads* and *premature writes*: delay both read and write operations
  - executions of transactions are called *strict* if both read and write operations on an object are delayed until all transactions that previously wrote that object have either committed or aborted.
  - the strict execution of transactions enforces the desired property of isolation
- Tentative versions are used during progress of a transaction
  - objects in tentative versions are stored in volatile memory