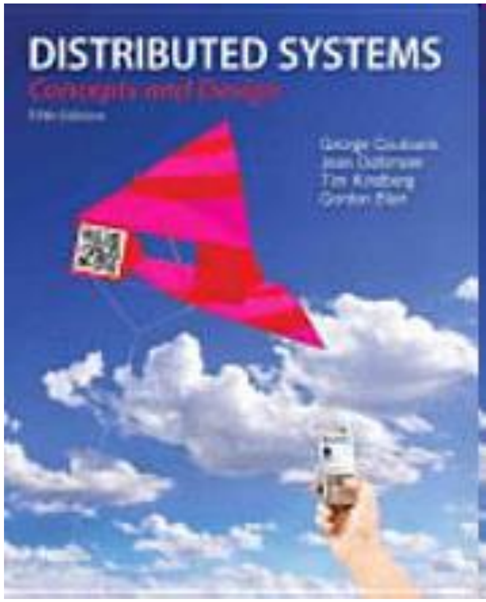
Distributed Systems Course

Transactions and Concurrency Control



16.1 Introduction

16.2 Transactions

16.3 Nested transactions

16.4 Locks

16.5 Optimistic concurrency control

16.6 Timestamp ordering

Definition of Transactions

- A transaction defines a sequence of server operations that is guaranteed by the server to be atomic in the presence of multiple clients and server crashes.

Introduction to transacti

File writes may fail

a) By writing nothing

b) By writing a wrong value, but checksums are used so that reads detect bad blocks

Therefore (a) and (b) are omission failures

Writing to the wrong block is an arbitrary failure.

- The goal of transacti
  - 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 (recovery in Chapter 17)
  - objects are stored in permanent storage
- Failure model

How can we deal with omission faults in disks?

  - 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

•

Operations of the *Account* interface

deposit(amount)
deposit amount in the account

withdraw(amount)
withdraw amount from the account

getBalance() $\rightarrow$ *amount*
return the balance of the account

setBalance(amount)
set the balance of the account to am

Used as an example. Each *Account* is represented by a remote object whose interface *Account* provides operations for making deposits and withdrawals and for setting and getting the balance.

Operations of the *Branch*

create(name) $\rightarrow$ *account*
create a new account with a given nam

lookUp(name) $\rightarrow$ *account*
return a reference to the account with the given name

branchTotal() $\rightarrow$ *amount*
return the total of all the balances at the branch

and each *Branch* of the bank is represented by a remote object whose interface *Branch* provides operations for creating a new account, looking one up by name and enquiring about the total funds at the branch. It stores a correspondence between account names and their remote object references

Atomic operations at server

- first we consider the synchronisation of client operations without 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)

Client cooperation by means of synchronizing server operations

- 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. one is a producer and another a consumer
  - e.g. one sets a lock and the other waits for it to be released
- 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)
- 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

Failure model for transactions

- Lampson's failure model deals with failures of disks, servers and communication.
  - 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

•

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
- Transactional file servers were built in the 1980s
- Transactions on distributed objects late 80s and 90s
- Middleware components e.g. CORBA Transaction service.
- Transactions apply to recoverable objects and are intended to be atomic.

Servers 'recover' - they are restarted and get their objects from permanent storage

A client's 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*

Atomicity of transactions

- The atomicity has two aspects

1. 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.

Concurrency control ensures isolation

2. 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

Failure atomicity and durability

- To support failure atomicity and durability, the objects must be recoverable.

1. When a server process crashes unexpectedly due to a hardware fault or software error, the changes due to all completed transactions must be available in permanent storage so that …
2. By the time a server acks the completion of a client's transaction, all of the transaction's changes to the objects must have been recorded in permanent storage.

Two ways for synchronization

- Perform the transactions serially

- Concurrency control

1. The aim for any server that supports transactions is to maximize concurrency.

2. Transactions are allowed to execute concurrently if they would have the same effect as a serial execution.

Operations in the *Coordinator* interface

- tra … er of re …
  - – … ct whose interface follows:

the client uses *OpenTransaction* to get *TID* from the coordinator

the client passes the *TID* with each request in the transaction

e.g. as an extra argument or transparently (The CORBA transaction service does uses 'context' to do this).

To commit - the client uses *closeTransaction* and the coordinator ensures that the objects are saved in permanent storage

starts a new transaction and delivers a unique TID *trans*. This … ion.

To abort - the client uses *abortTransaction* and the coordinator ensures that all temporary effects are invisible to other transactions

closeTransaction(trans) -> (commit, abort);

ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

The client asks either to commit or abort

aborts the transaction.

Transaction life histories

Why might a server abort a transaction?

because of concurrency control problems or if it crashes and then recovers

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

- 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?

the client finds out next time it tries to access an object at the server.

Concurrency control

- We will illustrate the 'lost update' and the 'inconsistent retrievals' problems which can occur in the absence of appropriate concurrency control
  - a lost update 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 retieval transaction observes values that are involved in an ongoing updating transaction
- we show how serial equivalent executions of transactions can avoid these problems
- we assume that the operations *deposit*, *withdraw*, *getBalance* and *setBalance* are *synchronized* operations - that is, their effect on the account balance is atomic.

The lost update problem

the net effect should be to increase *B* by 10% twice - 200, 220, 242.

but it only gets to 220. *T*'s update is lost.

Transaction ***T*** **:**		**Transaction** ***U*****:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *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%

The inconsistent retrievals proble

we see an inconsistent retrieval because *V* has only done the withdraw part when *W* sums balances of *A* and *B*

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	• •	

- The balances of A and B are both initially $200
- *V* transfers $100 from *A* to *B* while *W* calculates branch total (which should be $400 for account A and account B)

Serial equivalence

The transactions are scheduled to avoid overlapping access to the accounts accessed by both of them

- 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

A serially equivalent interleaving of *T* and *U* (lost updates cured)

their access to *B* is serial, the other part can overlap

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 can't get a lost update,
- the same is true if they are run in a serially equivalent ordering

A serially equivalent interleaving of *V* and *W* (inconsistent retri

we could overlap the first line of *W* with the second line of *V*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*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* runs 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

Read and *write* operation conflict rules

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

- 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)*

Serial equivalence defined in terms of conflicting operations

Which of their operations conflict?

T's write(i) conflicts with U's read (i)
U's read (j) and write(j) conflict with T's write(j)

- 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 and U access i and j

  - *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

- *T* accesses *i* before *U* and *T* accesses *j* before *U*. or
- U accesses *i* before *T* and *U* accesses j before *T.*

•Serial equivalence is used as a criterion for designing concurrency control schemes

A non-serially equivalent interleaving of operations of transactions *T* and *U*

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

- Each transaction's access to *i* and *j* is serialised with respect to one another, but
- *T* makes all accesses to *i* before *U* does
- *U* makes all accesses to *j* before *T* does
- therefore this interleaving is not serially equivalent

Recoverability from aborts

- if a transaction aborts, the server must make sure that other concurrent transactions do not see any of its effects
- we study two problems:
- ‘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
- (*getBalance* is a read operation and *setBalance* a write operation)

•

A dirty read when transaction *T* aborts

What is the problem?

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			

•*U* reads *A*'s balance (which was set by *T*) and then commits

T subsequently aborts.

U has performed a *dirty read*

These executions are serially equivalent

- *U* has committed, so it cannot be undone

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

What the potential problem?

cascading aborts

•

Cascading aborts

For recoverability - delay commits

- 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.
- Such situations are called *cascading aborts*.

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.

e.g. *U* waits to perform *getBalance* until *T* commits or aborts

Avoidance of cascading aborts is a stronger condition than recoverability

Premature writes - overwriting uncommitted values

Transaction *T*:		**Transaction *U*:**	
a.setBalance(105)		*a.setBalance(110)*	
	$100		
a.setBalance(105)	$105		
		a.setBalance(110)	$110

before *T* and *U* the balance of A was $100

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

Strict executions of transactions

- 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