Computer Science 425
Distributed Systems

CS 425 / CSE 424 / ECE 428

Fall 2012

Indranil Gupta (Indy)

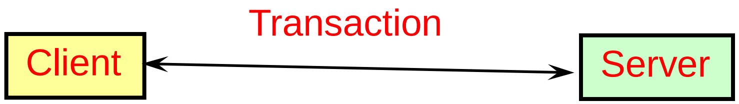
October 18, 2012

Lecture 16

Concurrency Control

Reading: Chapter 16.1,2,4 and 17.1,2,3,5 (relevant parts)

Transactions



Banking transaction for a customer (e.g., at ATM or browser)

Transfer $100 from saving to checking account;

Transfer $200 from money-market to checking account;

Withdraw $400 from checking account.

Transaction (invoked at client):	/* Every step is an RPC */
1. savings.withdraw(100)	/* includes verification */
2. checking.deposit(100)	/* depends on success of 1 */
3. mnymkt.withdraw(200)	/* includes verification */
4. checking. deposit(200)	/* depends on success of 3 */
5. checking.withdraw(400)	/* includes verification */
6. dispense(400)	
7. commit	

Bank Server: Coordinator *Interface*

- **All the following are RPCs from a client to the server**
- **Transaction calls that can be made at a client, and return values from the server:**

openTransaction() -> trans;

starts a new transaction and delivers a unique transaction identifier (TID) *trans*. This TID will be used in the other operations in the transaction.

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);

aborts the transaction.

- **TID can be passed implicitly (for other operations between open and close) with CORBA**

Bank Server: Account, Branch *interfaces*

Operations of the Account interface

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

Operations of the Branch interface

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

Transaction

- Sequence of operations that forms a single step, transforming the server data from one consistent state to another.
  - <u>All or nothing</u> principle: a transaction either completes successfully, and the effects are recorded in the objects, or it has no effect at all. (even with multiple clients, or crashes)
- A transactions is <u>indivisible</u> (atomic) from the point of view of other transactions
  - No access to intermediate results/states of other transactions
  - Free from interference by operations of other transactions

But…

- Transactions could run concurrently, i.e., with multiple clients
- Transactions may be distributed, i.e., across multiple servers

Transaction Failure Modes

Transaction:

1. savings.deduct(100)
2. checking.add(100)
3. mnymkt.deduct(200)
4. checking.add(200)
5. checking.deduct(400)
6. dispense(400)
7. commit

A failure at these points means the customer loses money; we need to restore old state

A failure at these points does not cause lost money, but old steps cannot be repeated

This is the point of no return

A failure after the commit point (ATM crashes) needs corrective action; no undoing possible.

Transactions in Traditional Databases (ACID)

- **Atomicity: All or nothing**
- **Consistency: if the server starts in a consistent state, the transaction ends the server in a consistent state.**
- **Isolation: Each transaction must be performed without interference from other transactions, i.e., the non-final effects of a transaction must not be visible to other transactions.**
- **Durability: After a transaction has completed successfully, all its effects are saved in permanent storage.**

  - **Atomicity: store tentative object updates (for later undo/redo) – many different ways of doing this**
  - **Durability: store entire results of transactions (all updated objects) to recover from permanent server crashes.**

Concurrent Transactions:Lost Update Problem

- **One transaction causes loss of info. for another: consider three account objects**

a: 100 b: 200 c: 300

Transaction T1	Transaction T2	
balance = b.getBalance()		
	balance = b.getBalance()	
	b.setBalance(balance*1.1)	b: 220
b.setBalance(balance*1.1)		b: 220
a.withdraw(balance* 0.1)		a: 80
	c.withdraw(balance*0.1)	c: 280

T1/T2's update on the shared object, "b", is lost

Conc. Trans.: Inconsistent Retrieval Prob.

❖ **Partial, incomplete results of one transaction are retrieved by another transaction.**

a: 100 b: 200 c: 300

Transaction T1		Transaction T2	total
a.withdraw(100)	a: 00		
		total = a.getBalance()	0.00
		total = total + b.getBalance	200
b.deposit(100)	b: 300		
		total = total + c.getBalance	500

T1's partial result is used by T2, giving the wrong result for T2

Concurrency Control: "Serial Equivalence"

- An interleaving of the operations of 2 or more transactions is said to be **serially equivalent** if the combined effect is the same as if these transactions had been performed sequentially (in some order).

a: 100 b: 200 c: 300

Transaction T1	Transaction T2
balance = b.getBalance()	
b.setBalance(balance*1.1)	
b: 220	
	balance = b.getBalance()
	b.setBalance(balance*1.1)
	b: 242
a.withdraw(balance* 0.1)	
a: 80	
	c.withdraw(balance*0.1)
	c: 278

== T1 (complete) followed by T2 (complete)

Checking Serial Equivalence – Conflicting Operations

- The effect of an operation refers to
  - The value of an object set by a write operation
  - The result returned by a read operation.
- Two operations are said to be conflicting operations, if their *combined effect* depends on the order they are executed, e.g., read-write, write-read, write-write (all on same variables). NOT read-read, NOT on different variables.
- *Two transactions are **serially equivalent** if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.*
  - *Why? Can start from original operation sequence and swap the order of non-conflicting operations to obtain a series of operations where one transaction finishes completely before the second transaction starts*
- Why is the above result important? Because: Serial equivalence is the basis for concurrency control protocols for transactions.

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

Concurrency Control: “Serial Equivalence”

❖ **An interleaving of the operations of 2 or more transactions is said to be serially equivalent if the combined effect is the same as if these transactions had been performed sequentially (in some order).**

a: 100   b: 200   c: 300

Transaction T1 | **Transaction T2**

== T1 (complete) followed by T2 (complete)

balance = b.getBalance()

b.setBalance(balance*1.1)

b: 220

balance = b.getBalance()

b.setBalance(balance*1.1)

b: 242

a.withdraw(balance* 0.1)

a: 80

c.withdraw(balance*0.1)

c: 278

Pairs of Conflicting Operations

Conflicting Operators Example

<u>Transaction T1</u>	<u>Transaction T2</u>	
x= a.read()		*Non-*serially equivalent interleaving of operations
a.write(20)		
	Conflicting Ops.	
	y = b.read()	
	b.write(30)	
b.write(x)		
	z = a.read()	

x= a.read()		Serially equivalent interleaving of operations
a.write(20)		
	z = a.read()	
b.write(x)		
	y = b.read()	
	b.write(30)	(why?)

Inconsistent Retrieval Prob – Caught!

❖ **Partial, incomplete results of one transaction are retrieved by another transaction.**

a: 100 b: 200 c: 300

Transaction T1		**Transaction T2**	total
a.withdraw(100)	a: 00		
		total = a.getBalance()	0.00
		total = total + b.getBalance	200
b.deposit(100)	b: 300		
		total = total + c.getBalance	500

T1's partial result is used by T2, giving the wrong result for T2

A Serially Equivalent Interleaving of T1 and T2

Transaction *T1*:		Transaction *T2*:	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	\$100		
		total = a.getBalance()	\$100
b.deposit(100)	\$300		
		total = total+b.getBalance()	\$400
		total = total+c.getBalance()	

Implementing Concurrent Transactions

- **How can we prevent isolation from being violated?**
- **Concurrent operations must be consistent:**
  - **If trans.T has executed a *read* operation on object A, a concurrent trans. U must not *write* to A until T commits or aborts.**
  - **If trans. T has executed a *write* operation on object A, a concurrent U must not *read or write* to A until T commits or aborts.**
- **How to implement this?**
  - **First cut: locks**

Example: Concurrent Transactions

- Exclusive Locks

Transaction T1

OpenTransaction()

balance = b.getBalance() Lock B

b.setBalance(balance*1.1)

a.withdraw(balance* 0.1) Lock A

CloseTransaction() UnLock B UnLock A

Transaction T2

OpenTransaction()

WAIT on B balance = b.getBalance()

…

…

Lock B

b.setBalance(balance*1.1)

c.withdraw(balance*0.1) Lock C

CloseTransaction() UnLock B UnLock C

Basic Locking

- Transaction managers (on server side) set locks on objects they need. A concurrent trans. cannot access locked objects.
- Two phase locking:
  - In the first (growing) phase of the transaction, new locks are only acquired, and in the second (shrinking) phase, locks are only released.
  - A transaction is not allowed acquire *any* new locks, once it has released any one lock.
- Strict two phase locking:
  - Locking on an object is performed only <u>before</u> the first request to read/write that object is about to be applied.
  - Unlocking is performed by the commit/abort operations of the transaction coordinator.
    - To prevent dirty reads and premature writes, a transaction waits for another to commit/abort
- However, use of separate read and write locks leads to more concurrency than a single exclusive lock – Next slide

2P Locking: Non-exclusive lock (per object)

non-exclusive lock compatibility

Lock already set	Lock requested: read	Lock requested: write
none	OK	OK
read	OK	WAIT
write	WAIT	WAIT

- A read lock is promoted to a write lock when the transaction needs write access to the same object.
- A read lock shared with other transactions' read lock(s) cannot be promoted. Transaction waits for other read locks to be released.
- Cannot demote a write lock to read lock during transaction – violates the 2P principle

Locking Procedure in Strict-2P Locking

- **When an operation accesses an object:**
  - **if you can promote a lock (nothing -> read -> write)**
  - **Don’t promote the lock if it would result in a conflict with another transaction’s already-existing lock**
    - **wait until all shared locks are released, then lock & proceed**
- **When a transaction <u>commits</u> or <u>aborts</u>:**
  - **release all locks that were set by the transaction**

Example: Concurrent Transactions

❖ Non-exclusive Locks

Transaction T1	Transaction T2
OpenTransaction()	
balance = b.getBalance() R-Lock B	
	OpenTransaction()
	balance = b.getBalance() R-Lock B
	b.setBalance(balance*1.1)
	Cannot Promote lock on B, Wait
Commit	
	Promote lock on B
	…

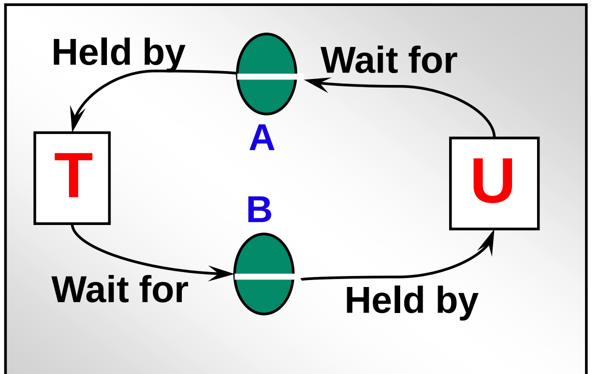
Example: Concurrent Transactions

- What happens in the example below?

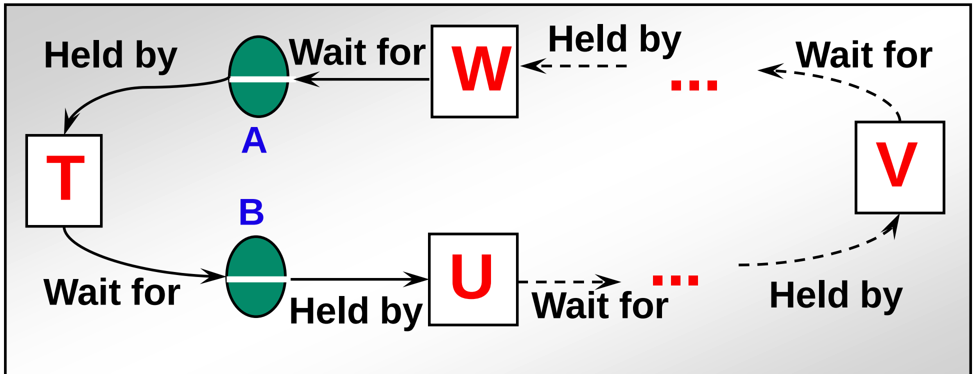
Transaction T1	Transaction T2
OpenTransaction()	
balance = b.getBalance() — R-Lock B	
	OpenTransaction()
	balance = b.getBalance() — R-Lock B
	b.setBalance(balance*1.1)
	Cannot Promote lock on B, Wait
b.setBalance=balance*1.1	
Cannot Promote lock on B, Wait	
…	…

Deadlocks

- Necessary conditions for deadlocks
  - Non-shareable resources (exclusive lock modes)
  - No preemption on locks
  - Hold & Wait or Circular Wait



Hold & Wait



Circular Wait

Naïve Deadlock Resolution Using Timeout

Transaction T		Transaction U	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
	(timeout elapses)	•••	
T's lock on *A* becomes vulnerable,	unlock *A*, abort T	•••	
		a.withdraw(200);	write locks *A* unlock *A*, *B*

Disadvantages?

Strategies to Fight Deadlock

- Lock timeout (costly and open to false positives)
- Deadlock Prevention: violate one of the necessary conditions for deadlock (from 2 slides ago), e.g., lock all objects before transaction starts, aborting if any fails
- Deadlock Avoidance: Have transactions declare max resources they will request, but allow them to lock at any time (Banker's algorithm)
- Deadlock Detection: detect cycles in the wait-for graph, and then abort one or more of the transactions in cycle

Summary

- Increasing concurrency important because it improves throughput at server (means more revenue $$$)
- Applications are willing to tolerate temporary inconsistency and deadlocks in turn
  - Need to detect and prevent these
- Driven and validated by actual application characteristics – mostly-read transactions abound

Midterm Statistics

- Graduate:
  - Count: 34
  - **Avg: 87.06**, StdDev: 11.3603275
  - Median: 91
  - **Max: 99 Min: 45**
- Undergraduate:
  - Count: 42
  - **Avg: 83.64**, StdDev: 10.79085301
  - Median: 87.75
  - **Max: 100 Min: 55**