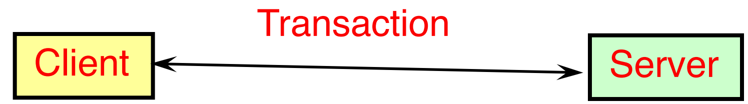
Distributed Systems

CS 425 / ECE 428

Transactions & Concurrency Control

Example Transaction



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):

1. savings.deduct(100) /* includes verification */
2. checking.add(100) /* depends on success of 1 */
3. mnymkt.deduct(200) /* includes verification */
4. checking.add(200) /* depends on success of 3 */
5. checking.deduct(400) /* includes verification */
6. dispense(400)
7. commit

Transaction

- A unit of work with the following properties
- Atomic – "all-or-nothing execution"
  - Two outcomes: commit or abort
- Consistent — takes server from one consistent state to another
- Isolated — does not interfere with other transactions
- Durable — effect of committed transaction persists after a crash (client or server)

Atomicity

Transaction

Lose money if these are split

1. savings.deduct(100)
2. checking.add(100)
3. mnymkt.deduct(200)
4. checking.add(200)

Leaves money in checking account

5. checking.deduct(400)
6. dispense(400)
7. commit

- **Whole transaction must be executed together**

Consistency

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

- Each account cannot have a negative balance
  - Must be true at the *end* of transaction
- Transaction aborted if consistency fails

Durability

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

- Result written in *durable* storage at commit time
  - Updates will persist even after server crash

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.

Bank Server: Coordinator *Interface*

❖ **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.

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

Properties of Transactions (ACID)

- **Atomicity: All or nothing**
- **Consistency: if the server starts in a consistent state, the transaction ends with 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 (we'll see them)**
  - **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

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)
a.withdraw(balance* 0.1)		
	a: 80	
		c.withdraw(balance*0.1)

b: 242

c: 278

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

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 in conflict, 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.**
- ***An execution of two transactions is 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.***

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

balance = b.getBalance()

b.setBalance = (balance*1.1)

a.withdraw(balance* 0.1)

Transaction T2

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

b: 220

balance = b.getBalance()

b.setBalance(balance*1.1)

b: 242

a: 80

c.withdraw(balance*0.1)

c: 278

Pairs of Conflicting Operations

Conflicting Operators Example

Transaction T1	Transaction T2	
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()	

Transaction T1	Transaction T2	
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 Retrievals Problem

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

Both withdraw and deposit contain a write operation

A Serially Equivalent Interleaving of V *and* W

Transaction *V*:		**Transaction *W*:**	
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

- Transaction operations can run concurrently, provided ACID is not violated, especially **isolation** principle
- 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			Transaction T2
OpenTransaction()			
balance = b.getBalance()	Lock B		OpenTransaction()
		WAIT on B	balance = b.getBalance()
b.setBalance = (balance*1.1)			...
a.withdraw(balance* 0.1)	Lock A		
CloseTransaction()	UnLock B UnLock A	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, 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 before 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 2P Locking

- **When an operation accesses an object:**
  - **if the object is not already locked, lock the object in the lowest appropriate mode & proceed.**
  - **if the object has a conflicting lock by another transaction, wait until object has been unlocked.**
  - **if the object has a non-conflicting lock by another transaction, share the lock & proceed.**
  - **if the object has a lower lock by the same transaction,**
    - **if the lock is not shared, promote the lock & proceed**
    - **else, wait until all shared locks are released, then lock & proceed**
- **When a transaction commits or aborts:**
  - **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	
…	…

Concurrent Transactions

- How many conflicts are there:
  - A: 0
  - B: 1
  - C: 2
  - D: 3
  - E: 4

T1:

a.read()
b.write()
c.read()
d.write()

T2:

c.read()
a.write()
d.read()
b.write()

Concurrent Transactions

T1:	T2:
a.read()	c.read()
b.write()	a.write()
c.read()	d.read()
d.write()	b.write()

- Is this a serially equivalent interleaving?
  - A: True
  - B: False

Concurrent Transactions

T1:	T2:
a.read()	
	c.read()
b.write()	
	a.write()
c.read()	
	d.read()
d.write()	
	b.write()

- Is this a serially equivalent interleaving?
    - A: True
    - B: False

Concurrent Transactions

T1:	T2:
a.read()	
	c.read()
b.write()	
	a.write()
c.read()	
d.write()	
	d.read()
	b.write()

- Is this a serially equivalent interleaving?
  - A: True
  - B: False

Why we need lock promotion

T1:

acquire R-lock on a
a.read()

release R-lock on a
acquire W-lock on a
a.write()
commit
release W-lock on a

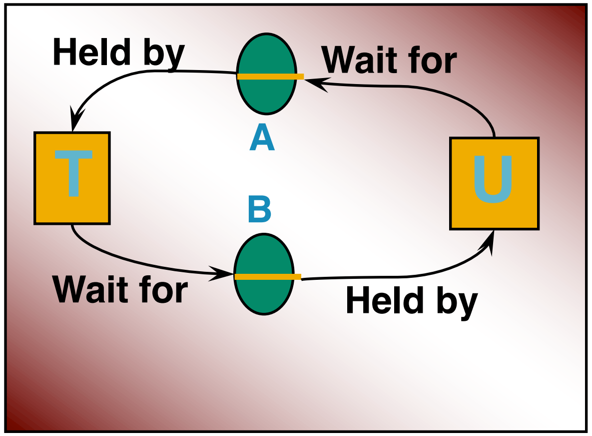
T2:

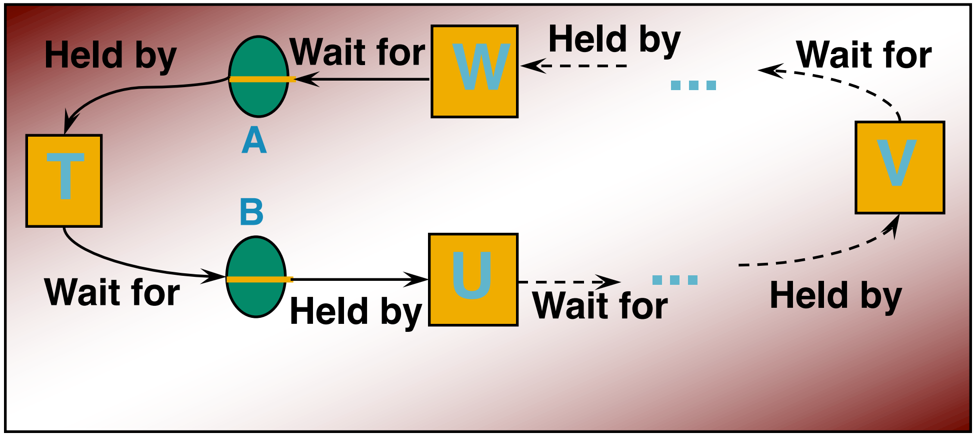
acquire R-lock on a
a.read()
release R-lock on a

acquire W-lock on a
a.write()
commit
release W-lock on a

Deadlocks

- Necessary conditions for deadlocks
  - Non-shareable resources (locked objects)
  - No preemption on locks
  - Hold & Wait
  - Circular Wait (Wait-for graph)





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); *commit*	write locks *a* unlock *a*, *b*

Deadlock Strategies

- Timeout
  - Too large -> long delays
  - Too small -> false positives
- Deadlock prevention
  - Lock all objects at transaction start
  - Use lock ordering
- Deadlock Detection
  - Maintain wait-for graph, look for cycle
  - Abort one transaction in cycle

Review Questions

- Which of the deadlock preconditions are violated by timeouts?
  - A: Exclusive access
  - B: No preemption
  - C: Hold-and-wait
  - D: Waits-for cycle

Review Questions

- Which of the deadlock preconditions are violated by lock ordering?
  - A: Exclusive access
  - B: No preemption
  - C: Hold-and-wait
  - D: Waits-for cycle

Review Questions

- If deadlocks are expected to occur frequently, which approach should we take?
  - A: Deadlock prevention
  - B: Deadlock detection
  - C: Timeouts

Concurrency control … summary so far …

- Increasing concurrency important because it improves throughput at server
- Applications are willing to tolerate temporary inconsistency and deadlocks in turn
- These inconsistencies and deadlocks need to be prevented or detected
- Driven and validated by actual application characteristics – mostly-read applications do not have too many conflicting operations anyway