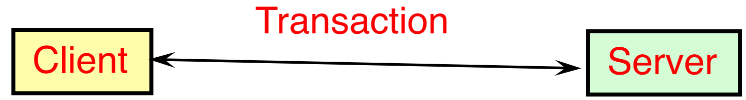
Distributed Systems

CS 425 / CSE 424 / ECE 428

Transactions & Concurrency Control

© 2010, I. Gupta, K. Nahrtstedt, S. Mitra, N. Vaidya, M. T. Harandi, J. Hou

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

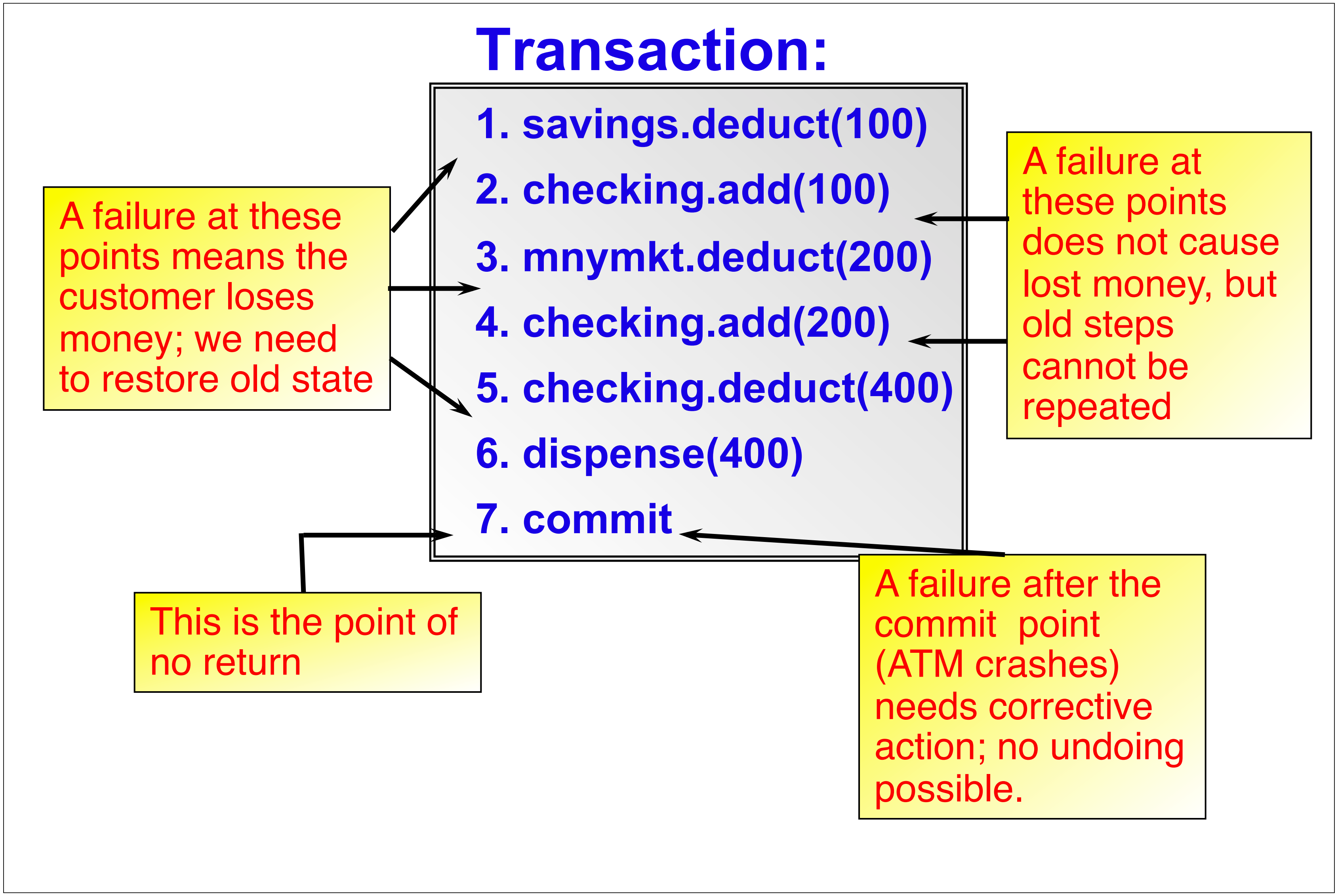
Transaction

- **Sequence of operations that forms a single step, transforming the server data from one consistent state to another.**
  - **All or nothing 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**
  - **Free from interference by other operations**

But…

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

Transaction Failure Modes



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.**

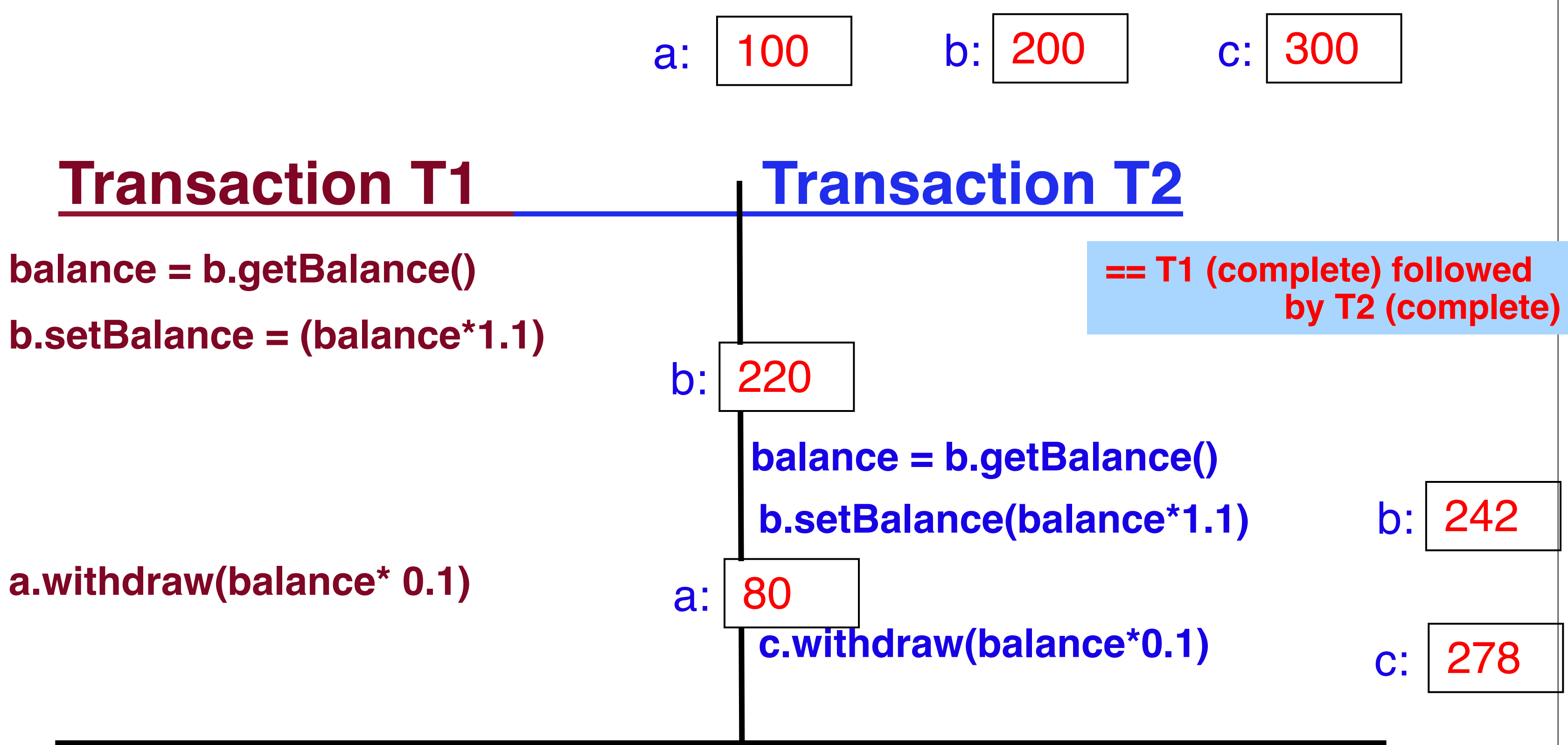
a: 100 b: 200 c: 300

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).



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.
- *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 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

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()	
a.write(20)	
	y = b.read()
	b.write(30)
b.write(x)	
	z = a.read()

Conflicting Ops.

*Non-*serially equivalent interleaving of operations

Transaction T1	Transaction T2
x= a.read()	
a.write(20)	
	z = a.read()
b.write(x)	
	y = b.read()
	b.write(30)

Serially equivalent interleaving of operations

(why?)

Inconsistent Retrievals Problem

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	:	

Both withdraw and deposit contain a write operation

A Serially Equivalent Interleaving of V and W

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()	

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

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, 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	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

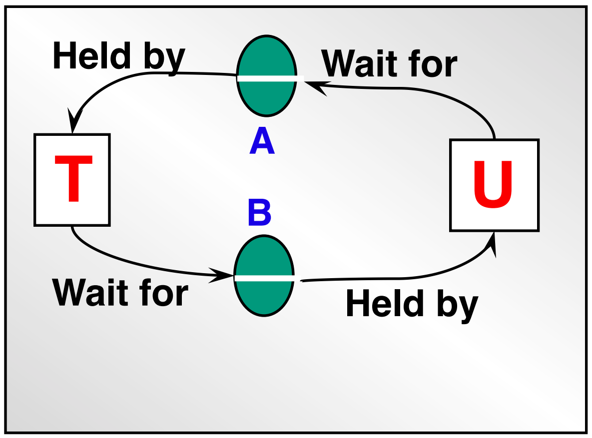
❖ What happens in the example below?

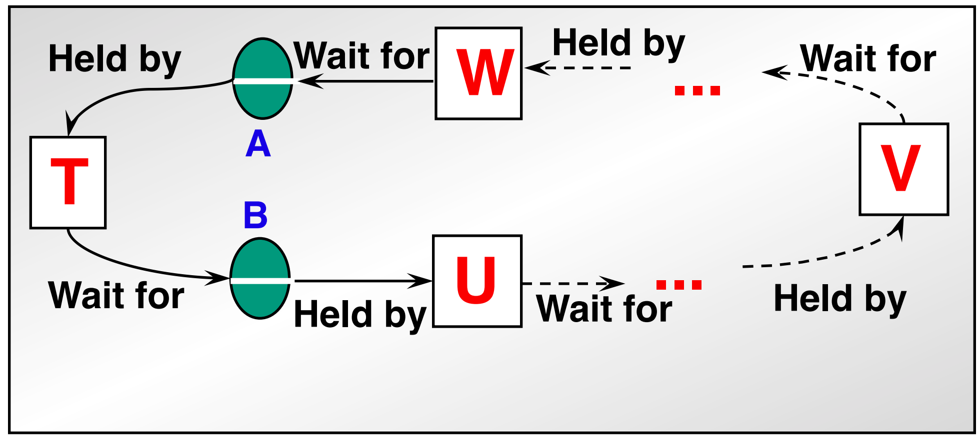
Transaction T1	Transaction T2
OpenTransaction()	
balance = b.getBalance() — R-Lock B	
	OpenTransaction()
	balance = b.getBalance() — R-Lock B
	b.setBalance =balance*1.1
	Cannot Promote lock on B, Wait
b.setBalance=balance*1.1	
Cannot Promote lock on B, Wait	
…	…

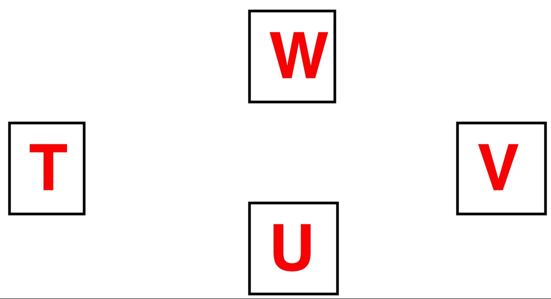
Deadlocks

❖ **Necessary conditions for deadlocks**

- ❑ **Non-shareable resources (locked objects)**
- ❑ **No preemption on locks**
- ❑ **Hold & Wait & Circular Wait (Wait-for graph)**







Complete this wait-for graph

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

- Deadlocks can be resolved by lock timeout (costly and open to false positives)
- Deadlock Prevention: violate one of the necessary conditions for deadlock (from previous slide), e.g., lock all objects at transaction start only; release all if any locking operation fails.
  Or, lock objects in a certain order (can force transactions to lock objects prematurely).
- Deadlock Detection: deadlocks can be detected, e.g., by using a wait-for graph, & then resolved by aborting one of the transactions in the cycle.

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