Bitcoin and Cryptocurrencies

- Lecture 1: Transactions
- Professor Radjabov Mukhammad

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

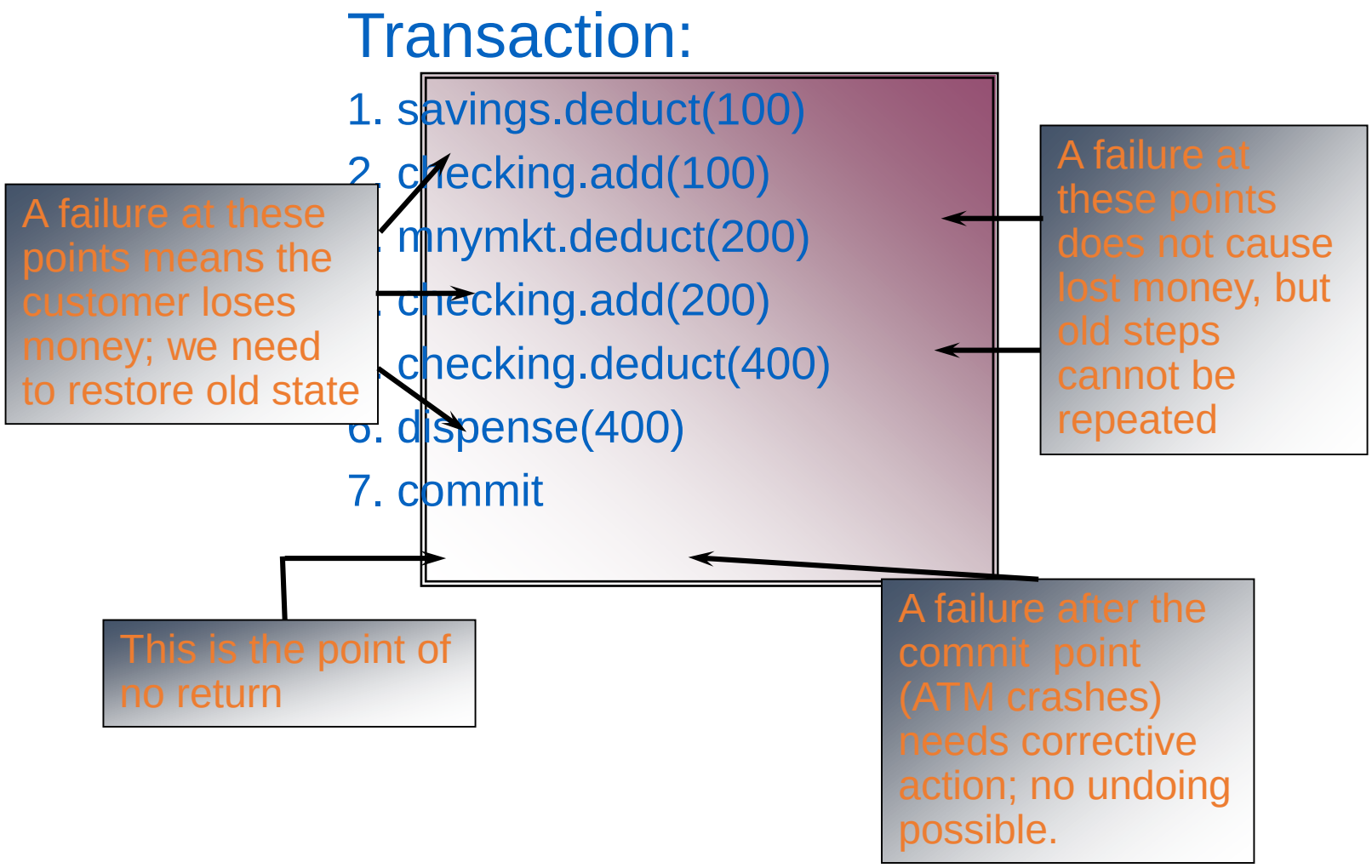
Transaction

- Sequence of operations that forms a single step, transforming the server data from one consistent state to another.
  - 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 indivisible (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



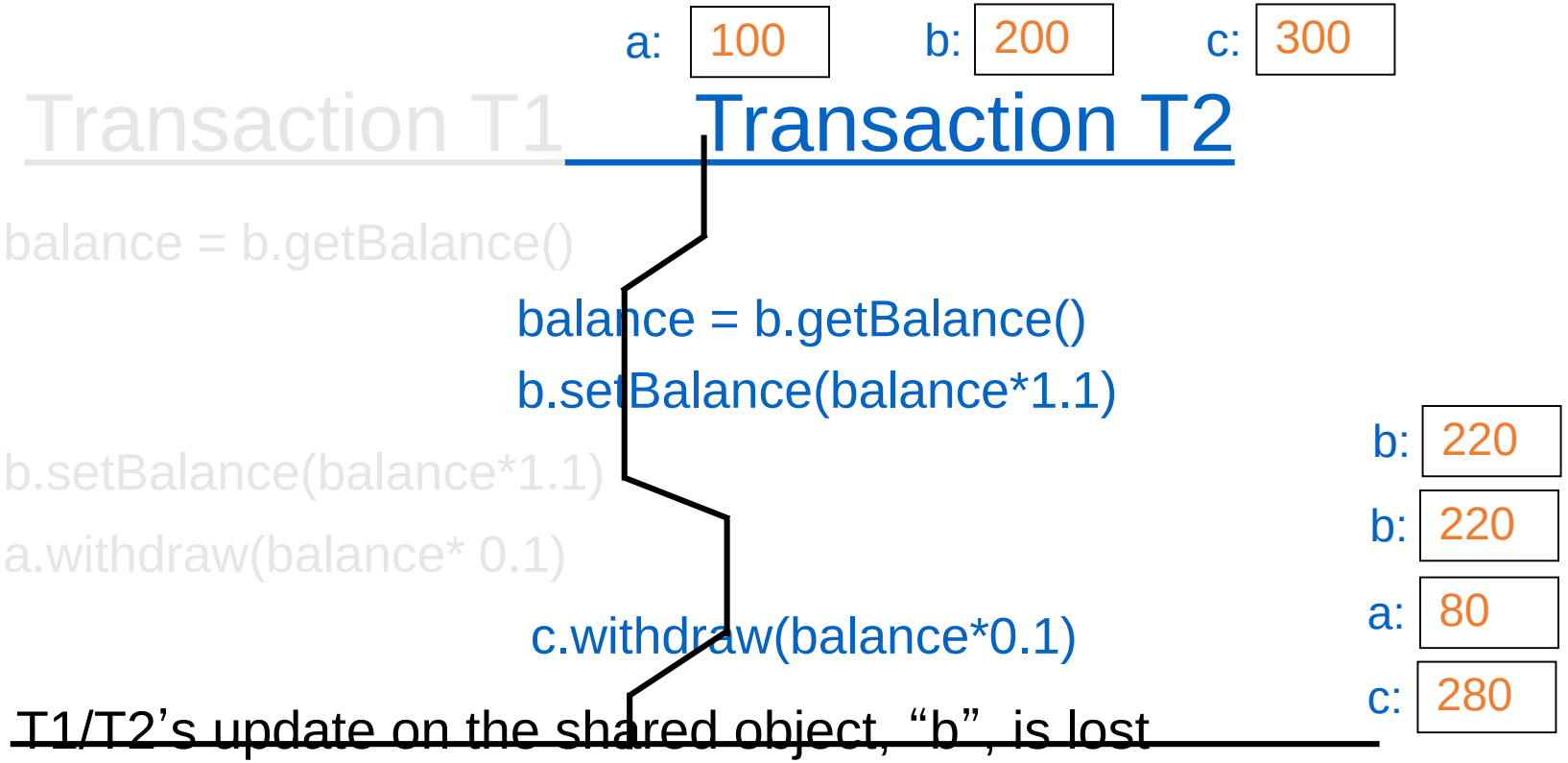
Transactions in Traditional Databases (ACID)

- 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.

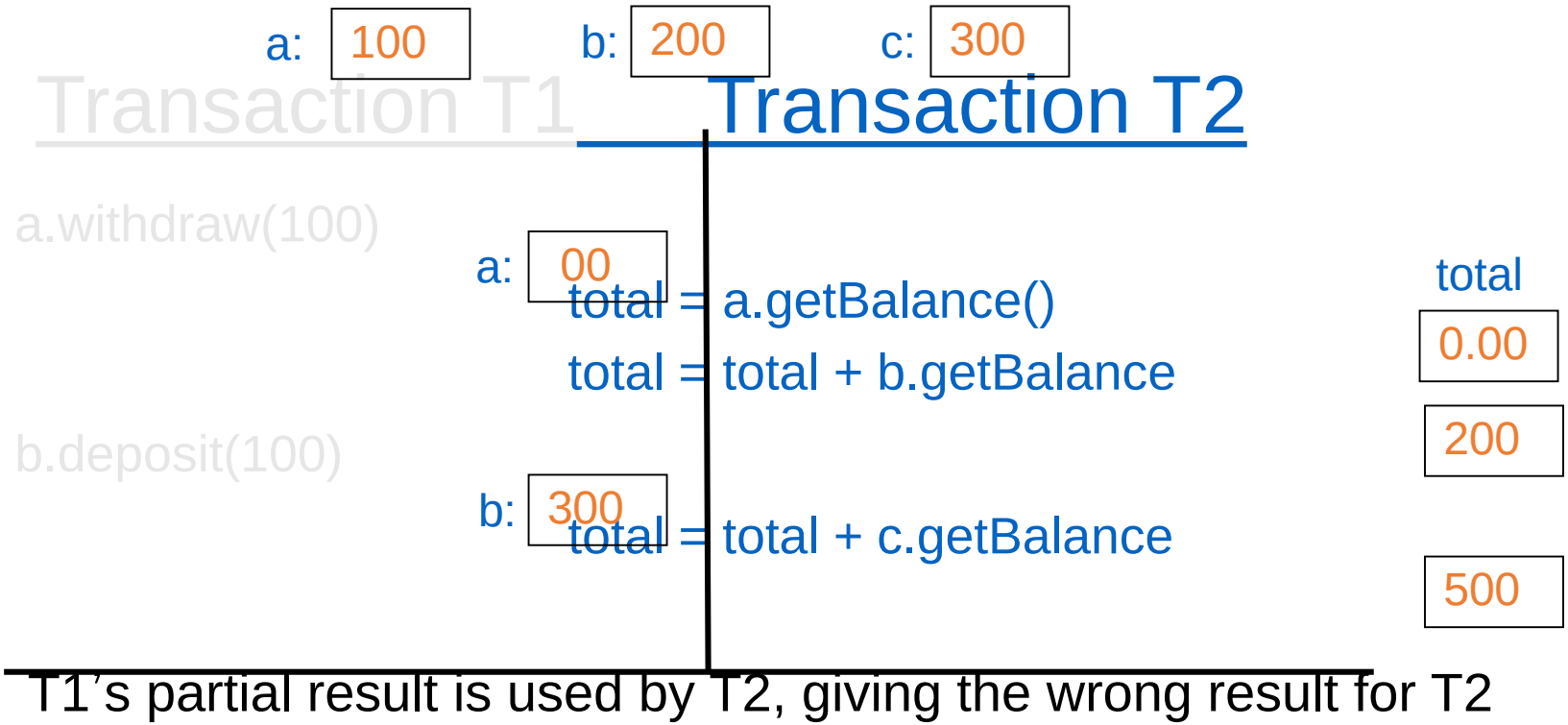
Concurrent Transactions:Lost Update Problem

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



Conc. Trans.: Inconsistent Retrieval Prob.

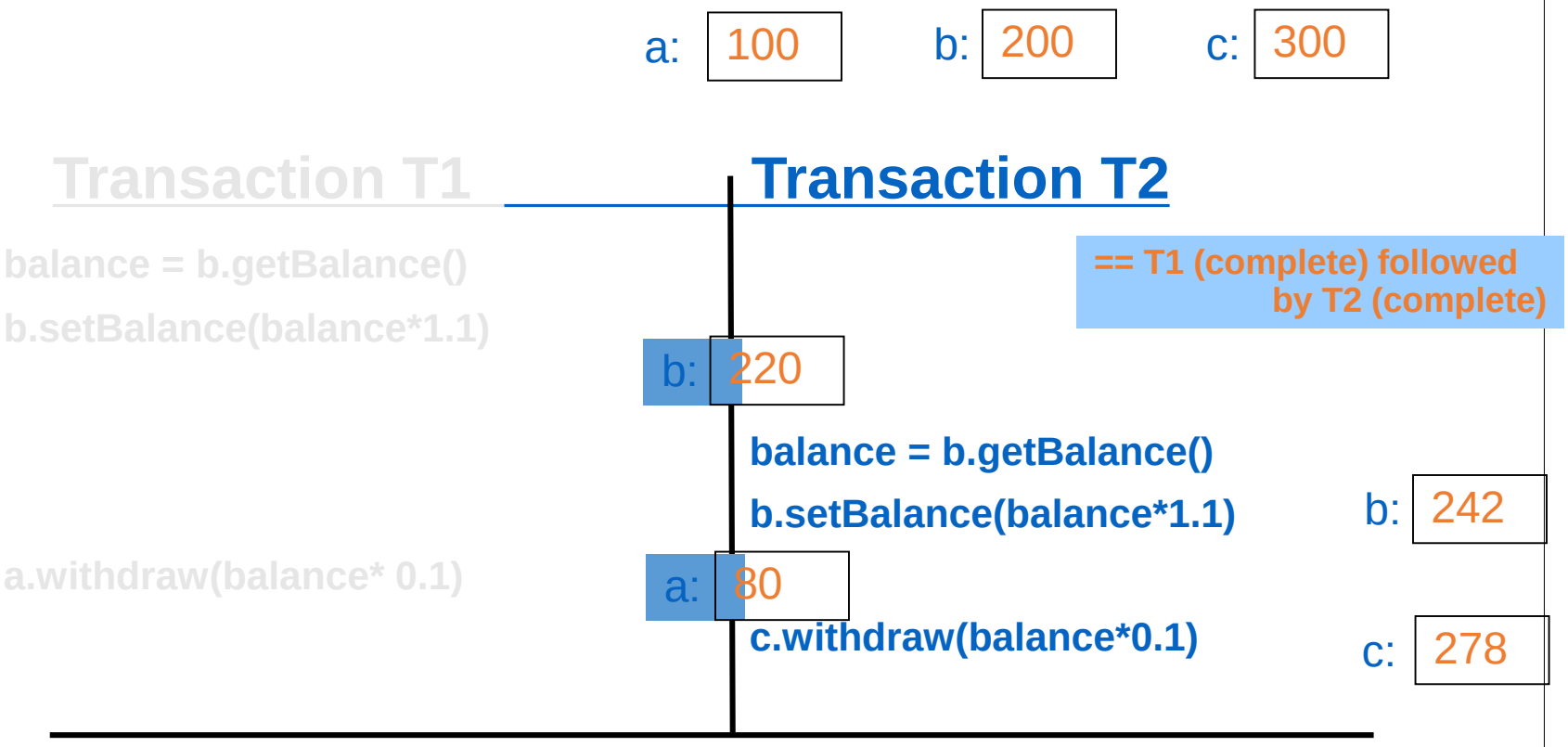
- Partial, incomplete results of one transaction are retrieved by another transaction.



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).**



Checking Serial Equivalence – Conflicting Operations

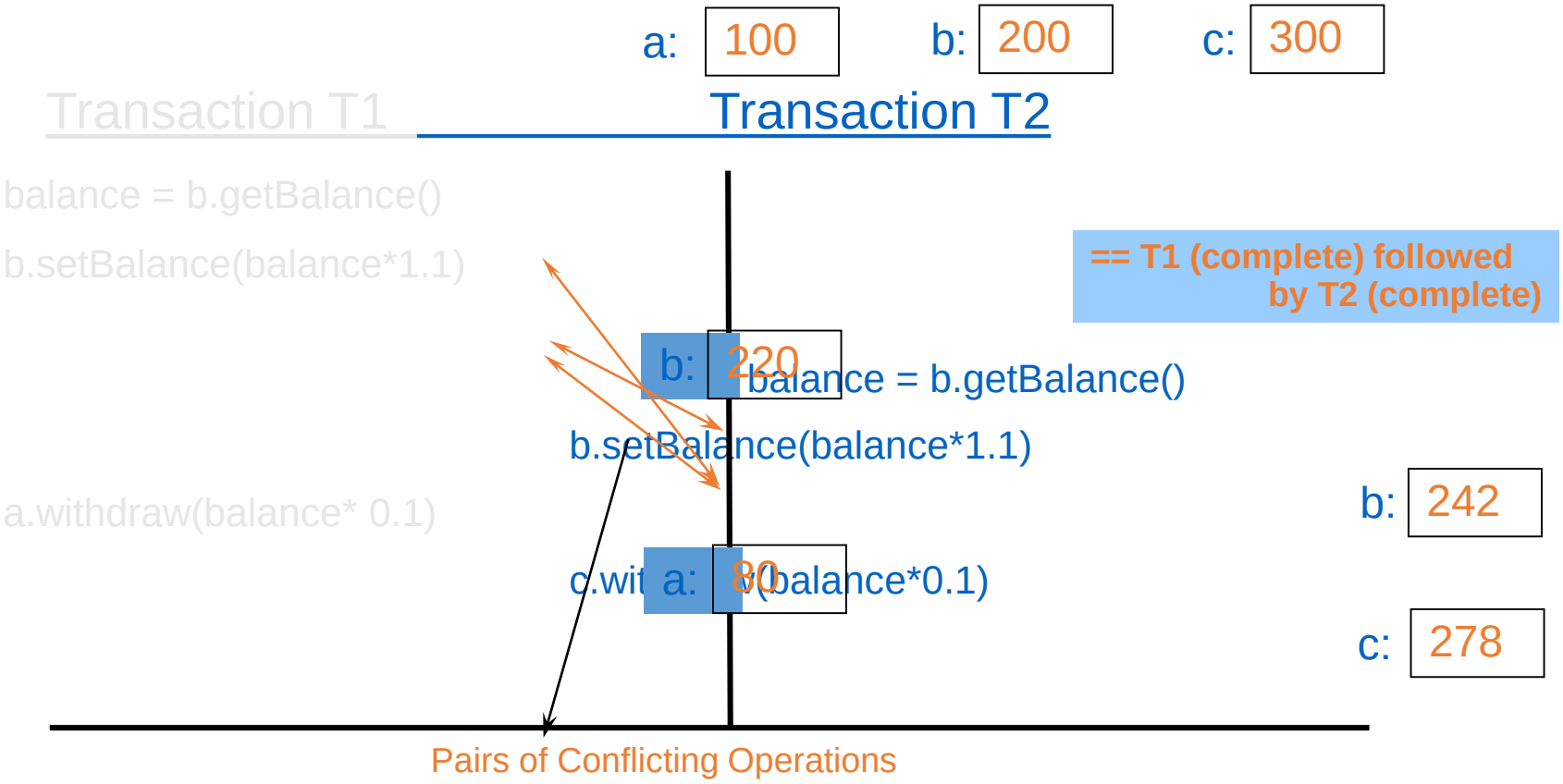
- The effect of an operation refers to
  - The value of an object set by a write operation
  - The result returned by a read operation.
- Two operations are said to be conflicting operations, if their *combined effect* depends on the order they are executed, e.g., read-write, write-read, write-write (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

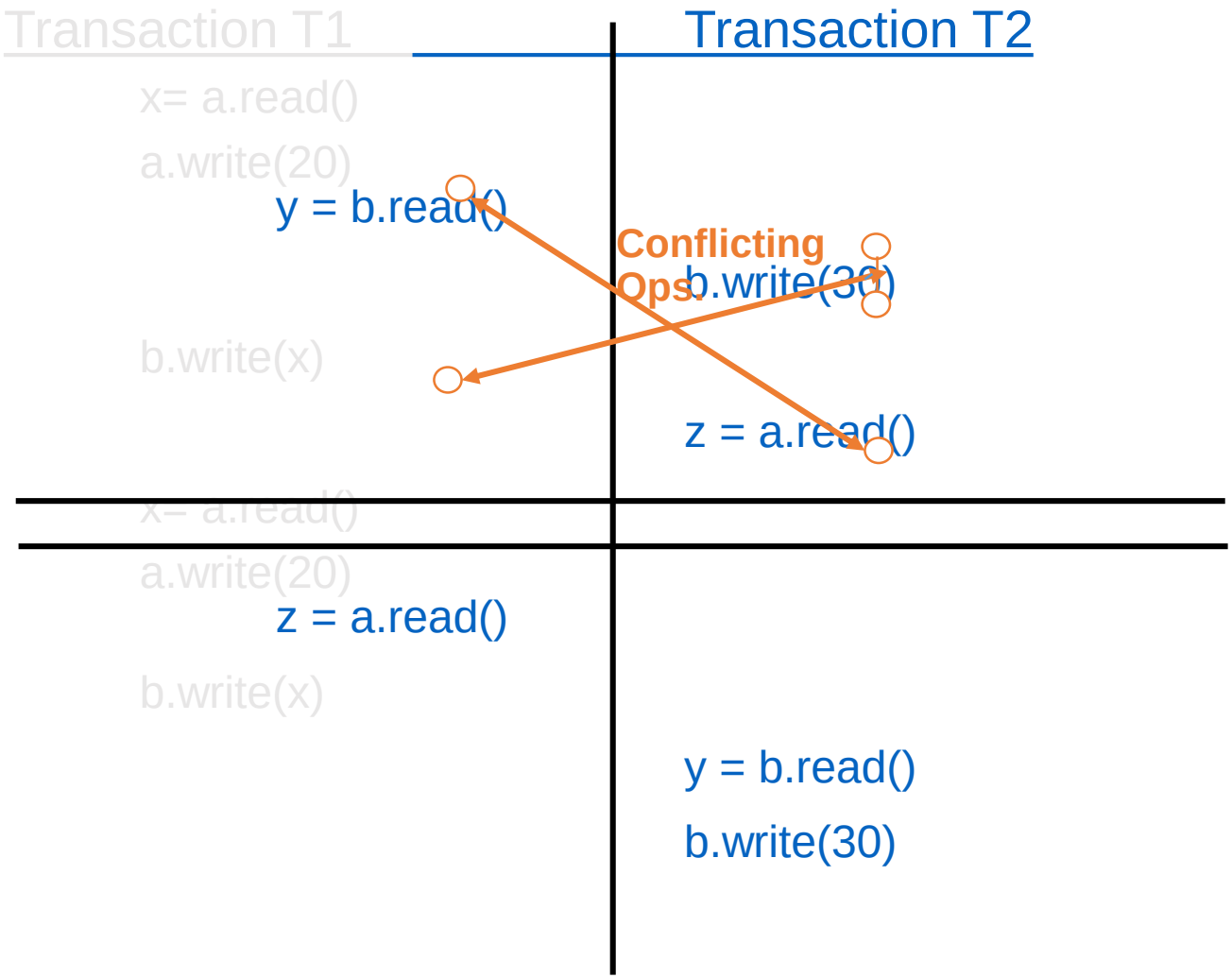
Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*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



Conflicting Operators Example



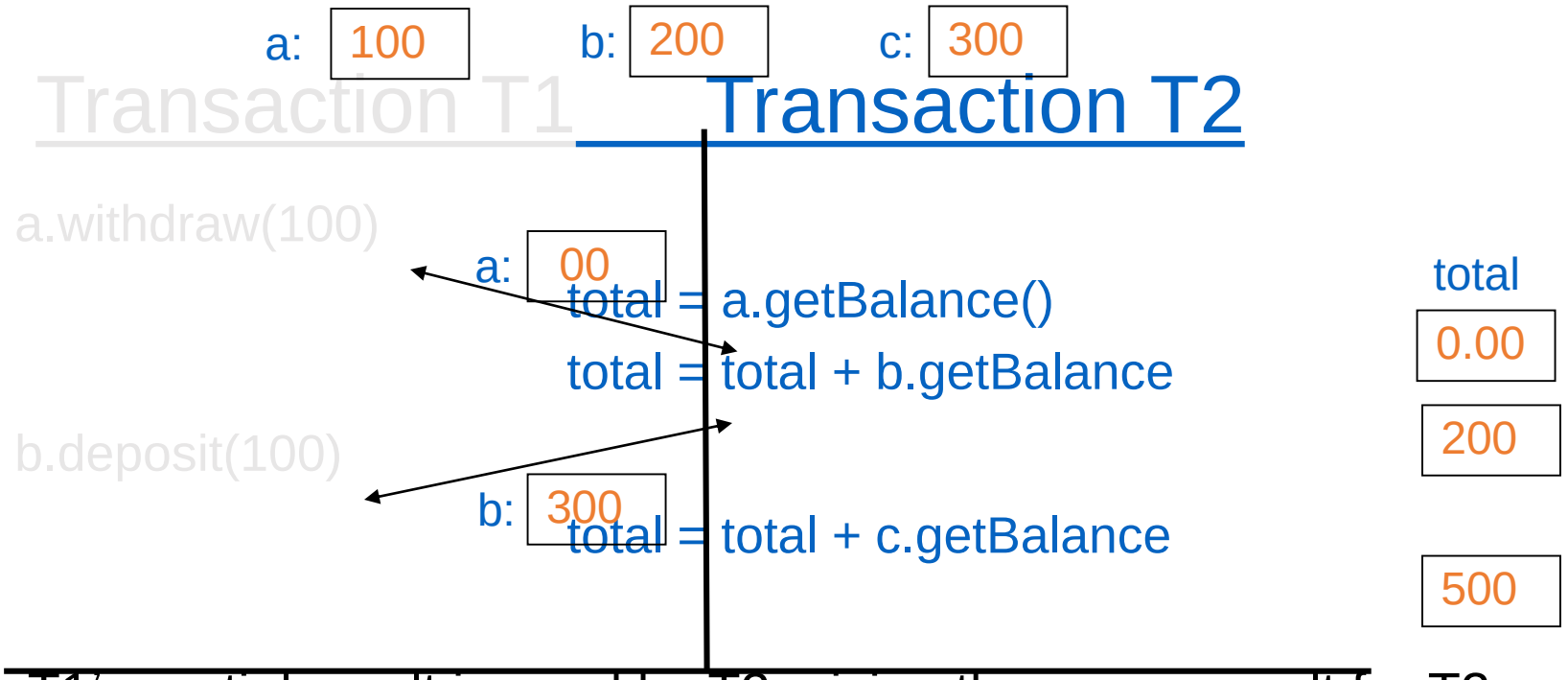
Non- serially equivalent interleaving of operations

Serially equivalent interleaving of operations

(why?)

Inconsistent Retrieval Prob

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



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		
b.deposit(100)	$300	*total = a.getBalance()*	$100
		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?

Concurrency control

- Lost update
  - 3 accounts (A, B, C)
    - with balances 100, 200, 300
  - T1 transfers from A to B, for 10% increase
  - T2 transfers from C to B, for 10% increase
  - Both T1, T2 read balance of B (200)
  - T1 overwrites the update by T2
    - Without seeing it

Transactions should not read a “stale” value & use it in computing a new value

Concurrency control

- Inconsistent retrievals
  - T1: transfers 10% of account A to account B
  - T2: computes sum of account balances
  - T2 computes sum before T1 updates B

Update transactions should not interfere with retrievals.

In general:
Transactions should not violate operation conflict rules.

Concurrency control

Serial equivalence
criterion for correct concurrent execution

T1 serially equivalent with T2 iff:

All pairs of conflicting operations of the two transactions are executed in the same order at all objects that both transactions access.

3 approaches to CC:
- Locking
- Optimistic CC
- Timestamp ordering

Tx's wait for one another
OR:
Restart Tx's after conflicts have been detected

Recoverability from aborts

- Servers must prevent a aborting Tx from affecting other concurrent Tx's.
  - Dirty reads:
    - T2 sees result update by T1 on account A
    - T2 performs its own update on A & then commits.
    - T1 aborts -> T2 has seen a "transient" value
      - T2 is not recoverable
    - If T2 delays its commit until T1's outcome is resolved:
      - Abort(T1) -> Abort(T2)
      - However, if T3 has seen results of T2:
        - Abort(T2) -> Abort(T3) !
  - Cascading aborts

Tx's should only read values written by committed Tx's

Recoverability from aborts

- Premature writes:
  - Assume server implements abort by maintaining the “before” image of all update operations
    - T1 & T2 both updates account A
    - T1 completes its work before T2
    - If T1 commits & T2 aborts, the balance of A is correct
    - If T1 aborts & T2 commits, the “before” image that is restored corresponds to the balance of A before T2
    - If both T1 & T2 abort, the “before” image that is restored corresponds to the balance of A as set by T1

Tx’s should be delayed until earlier Tx’s that update the Same objects have been either committed or aborted.

Recoverability from aborts

- Tx's should delay both their reads & updates in order to avoid interference
  - Strict execution -> enforce isolation
- Servers should maintain tentative versions of objects in volatile memory

Tx's should be delayed until earlier Tx's that update the Same objects have been either committed or aborted.

Concurrency Control: Locks

- Transactions:
  - Must be scheduled so that their effect on shared data is serially equivalent
  - Two types of approach
    - Pessimistic → If something can go wrong, it will
      Operations are synchronized before they are carried out
    - Optimistic → In general, nothing will go wrong
      Operations are carried out, synchronization at the end of the transaction
  - Locks (pessimistic)
    - can be used to ensuring serializability
    - lock(x), unlock(x)

Locks: Basics

- Oldest and most widely used CC algorithm
- A process before read/write ➔ requests the scheduler to grant a lock
- Upon finishing read/write ➔ the lock is released
- In order to ensure serialized transaction Two Phase Locking (2PL) is used

Locking

- How Locks prevent consistency problems
  - Lost update and inconsistent retrieval:
  - Causes:
    - are caused by the conflict between $r_i(x)$ and $w_j(x)$
    - two transactions read a value and use it to compute new value
  - Prevention:
    - delay the reads of later transactions until the earlier ones have completed

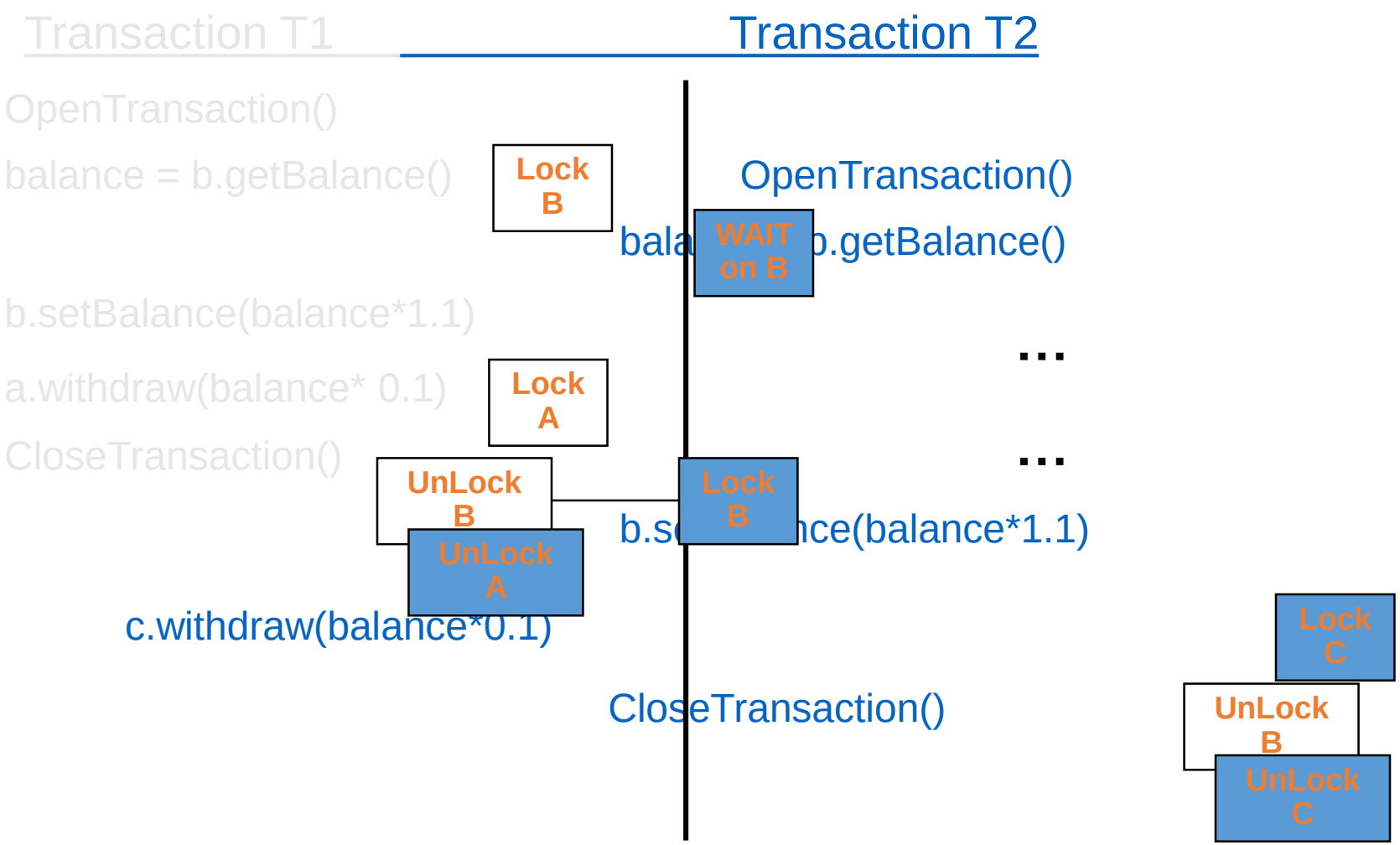
- Disadvantage of Locking
  - Deadlocks

2PL

- Strict 2PL avoids Cascading Aborts
  - A situation where a committed transaction has to be undone because it saw a file it shouldn’t have seen.
- Problems of Locking
  - Deadlocks
  - Livelocks
    - A transaction can’t proceed for an indefinite amount of time while other transactions continue normally. It happens due to unfair locking.
  - Lock overhead
    - If the system doesn’t allow shared access--wastage of resources
  - Avoidance of Cascading Aborts may be costly
    - Strict 2PL in fact, reduces the effect of concurrency

Example: Concurrent Transactions

- Exclusive Locks



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.

Basic Locking

- 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

Summary

- Increasing concurrency important because it improves throughput at server
- 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