Key Properties

Multiple computers
- Concurrent execution
- Independent failures
- Autonomous administrators
- Heterogeneous capacities, properties
- Large numbers (scalability)

Networked communication
- Asynchronous execution
- Unreliable delivery
- Insecure medium

Common goal
- Consistency – can discuss whole-system properties
- Transparency – can use the system without knowing details
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

Whole transaction must be executed together

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

Lose money if these are split

Leaves money in checking account
Consistency

Each account cannot have a negative balance
- Must be true at the end of transaction

Transaction aborted if consistency fails

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
Durability

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

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
Transaction Failure Modes

- A failure at these points means the customer loses money; we need to restore old state.
- A failure after the commit point (ATM crashes) needs corrective action; no undoing possible.
- A failure at these points does not cause lost money, but old steps cannot be repeated.

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

This is the point of no return.
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),module`;
  
  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()**
  - 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.
Lost Update Problem

- One transaction causes loss of info. for another:

  consider three account objects

```
Transaction T1
balance = b.getBalance()
b.setBalance = (balance*1.1)
a.withdraw(balance*0.1)
```

```
Transaction T2
balance = b.getBalance()
b.setBalance(balance*1.1)
```

T1/T2’s update on the shared object, “b”, is lost
Inconsistent Retrieval Prob.

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

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: withdraw(100)</td>
<td>a: 00</td>
</tr>
<tr>
<td>b: deposit(100)</td>
<td>b: 300</td>
</tr>
<tr>
<td>a: 100</td>
<td>b: 200</td>
</tr>
<tr>
<td>total = a.getBalance()</td>
<td>total = total + b.getBalance</td>
</tr>
<tr>
<td>total = total + b.getBalance</td>
<td></td>
</tr>
<tr>
<td>T1’s partial result is used by T2, giving the wrong result</td>
<td></td>
</tr>
</tbody>
</table>

a: 100 b: 200 c: 300

100 200 300

0 200 500

0 200 500
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).

**Serial Equivalence**

---

Transaction T1

- balance = b.getBalance()
- b.setBalance = (balance*1.1)
- a.withdraw(balance*0.1)

Transaction T2

- balance = b.getBalance()
- b.setBalance(balance*1.1)
- c.withdraw(balance*0.1)

---

**Example:**

- 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

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read read</td>
<td>No</td>
<td>Because the effect of a pair of read operations does not depend on the order in which they are executed</td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
<td>Because the effect of a read and a write operation depends on the order of their execution</td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
<td>Because the effect of a pair of write operations depends on the order of their execution</td>
</tr>
</tbody>
</table>
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).

Transaction T1

- balance = b.getBalance()
- b.setBalance = (balance*1.1)
- a.withdraw(balance*0.1)

Transaction T2

- balance = b.getBalance()
- b.setBalance(balance*1.1)
- c.withdraw(balance*0.1)

Pairs of Conflicting Operations

- T1 (complete) followed by T2 (complete)
Example

Transaction T1

- x = a.read()
- a.write(20)
- b.write(x)

Transaction T2

- y = b.read()
- b.write(30)
- z = a.read()

Two operations overlap. This is called

**Serially equivalent interleaving of operations** (why?)

**Non-serially equivalent interleaving of operations**

- x = a.read()
- a.write(20)
- b.write(x)

- z = a.read()
- y = b.read()
- b.write(30)

(why?)
Inconsistent Retrievals Problem

Transaction V:
\[ a.\text{withdraw}(100) \]
\[ b.\text{deposit}(100) \]

Transaction W:
\[ a\text{Branch}.\text{branchTotal}() \]

\[ a.\text{withdraw}(100); \quad \$100 \]

\[ a.\text{withdraw}(100); \quad total = a.\text{getBalance}() \quad \$100 \]

\[ total = total + b.\text{getBalance}() \quad \$300 \]

\[ total = total + c.\text{getBalance}() \]

\[ b.\text{deposit}(100) \quad \$300 \]

Both withdraw and deposit contain a write operation.
## A Serially Equivalent Interleaving of V and W

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.withdraw(100);</code></td>
<td><code>aBranch.branchTotal()</code></td>
</tr>
<tr>
<td><code>b.deposit(100)</code></td>
<td></td>
</tr>
<tr>
<td><code>a.withdraw(100);</code></td>
<td><code>total = a.getBalance()</code></td>
</tr>
<tr>
<td><code>b.deposit(100)</code></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$100$</td>
<td><code>$100$</code></td>
</tr>
<tr>
<td>$300$</td>
<td><code>$400$</code></td>
</tr>
</tbody>
</table>

```
Transaction V:
a.withdraw(100);
b.deposit(100)
```

```
Transaction W:
aBranch.branchTotal()
```

```
total = a.getBalance() + b.getBalance() + 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
Exclusive Locks

Transaction T1

OpenTransaction()

balance = b.getBalance()

b.setBalance = (balance*1.1)

a.withdraw(balance*0.1)

CloseTransaction()

Transaction T2

OpenTransaction()

balance = b.getBalance()

... 

b.setBalance = (balance*1.1)

c.withdraw(balance*0.1)

CloseTransaction()
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)

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

<table>
<thead>
<tr>
<th>Lock set</th>
<th>Lock requested</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Read</td>
<td>OK</td>
</tr>
<tr>
<td>Read</td>
<td>Write</td>
<td>Wait</td>
</tr>
<tr>
<td>Write</td>
<td>Read</td>
<td>Wait</td>
</tr>
<tr>
<td>Write</td>
<td>Write</td>
<td>Wait</td>
</tr>
</tbody>
</table>
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

OpenTransaction()

balance = b.getBalance()

Commit

Transaction T2

OpenTransaction()

balance = b.getBalance()

b.setBalance = balance*1.1

Cannot Promote lock on B, Wait

Promote lock on B

...
Example: Concurrent Transactions

What happens in the example below?

Transaction T1

OpenTransaction()
balance = b.getBalance()
b.setBalance = balance*1.1

Transaction T2

OpenTransaction()
balance = b.getBalance()
b.setBalance = balance*1.1

...
Concurrent Transactions

- How many conflicts are there:
  - T1:
    - a.read()
    - b.write()
    - c.read()
    - d.write()
  - T2:
    - c.read()
    - a.write()
    - d.read()
    - b.write()
Concurrent Transactions

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

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

• Is this a serially equivalent interleaving?
  • A: True
  • B: False
Concurrent Transactions

T₁:

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

T₂:

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

• Is this a serially equivalent interleaving?
Concurrent Transactions

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

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

- Is this a serially equivalent interleaving?
Why we need lock promotion

T1:

\textit{acquire R-lock on a}
\textit{a.read()}
\textit{release R-lock on a}
\textit{acquire W-lock on a}
\textit{a.write()}
\textit{commit}
\textit{release W-lock on a}

T2:

\textit{acquire R-lock on a}
\textit{a.read()}
\textit{release R-lock on a}
\textit{acquire W-lock on a}
\textit{a.write()}
\textit{commit}
\textit{release W-lock on a}
Necessary conditions for deadlocks
- Non-shareable resources (locked objects)
- No preemption on locks
- Hold & Wait
- Circular Wait (Wait-for graph)
## Deadlock Resolution Using Timeout

<table>
<thead>
<tr>
<th>Operation</th>
<th>Locks</th>
<th>Operation</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transaction T</strong></td>
<td></td>
<td><strong>Transaction U</strong></td>
<td></td>
</tr>
<tr>
<td>Operations</td>
<td>Locks</td>
<td>Operations</td>
<td>Locks</td>
</tr>
<tr>
<td><code>a.deposit(100);</code></td>
<td>write lock a</td>
<td><code>b.deposit(200)</code></td>
<td>write lock b</td>
</tr>
<tr>
<td><code>b.withdraw(100)</code></td>
<td>wait for U’s lock on b</td>
<td><code>a.withdraw(200);</code></td>
<td>waits for T’s lock on a</td>
</tr>
<tr>
<td></td>
<td>(timeout elapses)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T’s lock on A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>becomes vulnerable,</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock a, abort T</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>a.withdraw(200);</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>commit</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>unlock a, b</code></td>
<td></td>
</tr>
</tbody>
</table>
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
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