Transactions

- Transactions provide higher level mechanism for *atomicity of* processing in distributed systems
  - Have their origins in databases
- Banking example: Three accounts A:$100, B:$200, C:$300
  - Client 1: transfer $4 from A to B
  - Client 2: transfer $3 from C to B
- Result can be inconsistent unless certain properties are imposed on the accesses

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A: $100</td>
<td>Read C: $300</td>
</tr>
<tr>
<td>Write A: $96</td>
<td>Write C: $297</td>
</tr>
<tr>
<td>Read B: $200</td>
<td>Read B: $200</td>
</tr>
<tr>
<td>Write B: $204</td>
<td>Write B: $203</td>
</tr>
</tbody>
</table>

ACID Properties

- *Atomicity*: all or nothing
- *Consistency*: transaction takes system from one consistent state to another
- *Isolation*: immediate effects are not visible to other transactions (serializability)
- *Durability*: changes are permanent once transaction completes (commits)

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A: $100</td>
<td>Read C: $300</td>
</tr>
<tr>
<td>Write A: $96</td>
<td>Write C: $297</td>
</tr>
<tr>
<td>Read B: $200</td>
<td>Read B: $204</td>
</tr>
<tr>
<td>Write B: $204</td>
<td>Write B: $207</td>
</tr>
</tbody>
</table>
Transaction Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END_TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

Example: airline reservation

Begin_transaction
if(reserve(NY, Paris)==full) Abort_transaction
if(reserve(Paris, Athens)==full) Abort_transaction
if(reserve(Athens, Hong Kong)==full) Abort_transaction
End_transaction

Distributed Transactions

- Nested transactions: allow transactions to be composed of other transactions
  - The outermost transaction is called the top-level transaction, other transactions are called subtransactions
  - Subtransactions at the same level may run concurrently.
- A distributed transaction is a transaction that accesses objects managed by multiple servers
  - Nested transactions are useful in distributed systems because subtransactions may run concurrently in different servers
Implementation: Private Workspace

- Each transaction gets its own copy of all objects
  - Can optimize for reads by not making copies
  - Can optimize for writes by copying only what is required
- Commit requires making local workspace global
- Abort: throw away the private copy

Option 2: Write-ahead Logs

- *In-place updates*: transaction makes changes directly to all objects
- *Write-ahead log*: prior to making change, transaction writes to log on *stable storage*
  - Transaction ID, object ID, original value, new value
- If transaction commits:
  - Write a *commit* record onto the log
- If transaction aborts
  - Read log records and undo changes [rollback]
Write-ahead Log Example

\[
\begin{array}{|c|c|c|c|}
\hline
& Log & Log & Log \\
\hline
x = 0; & [x = 0 / 1] & [x = 0 / 1] & [x = 0 / 1] \\
y = 0; & [y = 0 / 2] & [y = 0 / 2] & [y = 0 / 2] \\
BEGIN_TRANSACTION; & & & \\
x = x + 1; & & & \\
y = y + 2 & & & \\
x = y * y; & (a) & (b) & (c) & (d) \\
END_TRANSACTION; & & & \\
\hline
\end{array}
\]

a) A transaction.
b) The log before each statement is executed.

Concurrency Control

• Goal: Allow several transactions to be executing simultaneously such that
  – Collection of manipulated data items is left in a consistent state
• Achieve consistency by ensuring data items are accessed in a specific order
  – Final result should be the same as if the transactions ran sequentially
Concurrency Control Implementation

- General organization of managers for handling transactions.

Serializability

- Key idea: properly schedule conflicting operations
- A pair of operations conflicts if at least one operation is write
  - Read-write conflict
  - Write-write conflict
Optimistic Concurrency Control

• Transaction does what it wants and validates changes prior to commit
  – Check if any object has been changed by committed transactions since it was opened
  – Insight: conflicts are rare, so works well most of the time
• Advantage:
  – Deadlock free
  – Maximum parallelism
• Disadvantage:
  – Rerun transaction if aborts
  – Probability of conflict rises substantially at high loads
• Not used widely

Two-Phase Locking (2PL)

• Widely used concurrency control technique
• Scheduler acquires all necessary locks in growing phase, releases locks in shrinking phase
  – Check if operation on data item x conflicts with existing locks
    • If so, delay transaction. If not, grant a lock on x
  – Never release a lock until data manager finishes operation on x
  – Once a lock is released, no further locks can be granted
• 2PL guarantees serializability
• Problem: deadlock possible
  – Solution: order objects, transaction can only request locks in order of the objects
Two-Phase Locking

- Problem with 2PL: other transactions can access data written by a uncommitted transaction
  - Cascading aborts possible
- Solution: locks can only be released when the transaction commits

Strict Two-Phase Locking
Timestamp-based Concurrency Control

- Each transaction $T_i$ is given timestamp $ts(T_i)$ when it starts.
- If $T_i$ wants to do an operation that conflicts with $T_j$:
  - Abort $T_i$ if $ts(T_i) < ts(T_j)$.
- When a transaction aborts, it must restart with a new (larger) timestamp.
- Two values for each data item $x$:
  - $Max-rts(x)$: max timestamp of a transaction that read $x$.
  - $Max-wts(x)$: max timestamp of a transaction that wrote $x$.

Reads and Writes using Timestamps

- $Read_i(x)$
  - If $ts(T_i) < max-wts(x)$ then Abort $T_i$.
  - Else
    - Perform $R_i(x)$.
    - $Max-rts(x) = max(max-rts(x), ts(T_j))$.
- $Write_i(x)$
  - If $ts(T_j) < max-rts(x)$ or $ts(T_i) < max-wts(x)$ then Abort $T_i$.
  - Else
    - Perform $W_i(x)$.
    - $Max-wts(x) = ts(T_j)$. 