Transactions and Concurrency Control

Concurrency

**Concurrency** noun \kən-ˈkər-ən(t)-sē\  the simultaneous occurrence of events or circumstances

- Happens at various scales
- Enables efficient use of resources
A Christmas example

Your grandmother puts 1000 SEK in your account.

\[ y_1 = \text{readBalance}(Y) \]
\[ g = \text{readBalance}(G) \]
\[ g = g - 1000 \]
\[ y_1 = y_1 + 1000 \]
\[ \text{writeBalance}(Y, y_1) \]
\[ \text{writeBalance}(G, g) \]

Your uncle puts 500 SEK in your account.

\[ y_2 = \text{readBalance}(Y) \]
\[ u = \text{readBalance}(U) \]
\[ u = u - 500 \]
\[ y_2 = y_2 + 500 \]
\[ \text{writeBalance}(Y, y_2) \]
\[ \text{writeBalance}(U, u) \]

Uncontrolled concurrency

- What if both execute the transfer at the same time?
- Operations of the transactions are **interleaved**

\[ y_1 = \text{readBalance}(Y) \]
\[ g = \text{readBalance}(G) \]
\[ g = g - 1000 \]
\[ y_1 = y_1 + 1000 \]
\[ \text{writeBalance}(Y, y_1) \]
\[ \text{writeBalance}(G, g) \]

\[ y_2 = \text{readBalance}(Y) \]
\[ u = \text{readBalance}(U) \]
\[ u = u - 500 \]
\[ y_2 = y_2 + 500 \]
\[ \text{writeBalance}(Y, y_2) \]
\[ \text{writeBalance}(U, u) \]

Oops!

Your account is only credited 500 SEK!
Dealing with concurrency

- Uncontrolled concurrency can (and has) lead to severe problems
- Problems often hard to reproduce
- Understanding concurrency is a crucial skill for any designer of a computer

Outline

- Motivation
- Transactions & ACID properties
  - Isolation
    - Conflicts & serializability
    - 2-phase locking
    - Deadlocks
- Atomicity
  - 2-phase commit protocol
**Transactions**

- **Definition:** A **transaction** is a **sequence of operations** that must to be treated as an undivided **unit**.
- Operations read and write
  - `read(X)`: access resource X without modifying it
  - `write(X)`: modify resource X
- A transaction must be either **committed** or **aborted**

**ACID properties**

- Desirable properties for transactions
  - **Atomicity**
  - **Consistency**
  - **Isolation**
  - **Durability**
Atomicity

- “Either all or none of the effects a transaction are applied”

- Ancient greek ἄτομος (atomos): indivisible
- If a transaction commits, all of its effects are visible
- If a transaction aborts, none of its effects are visible

- After reboot of server, modifications of X must not be visible!

Consistency

- “Transactions maintain any internal invariant”

- Invariant: a property of a system that always holds true
- Consistent ⇔ all invariants hold

Transaction would violate the invariant “Balance on G must be larger than 0.”
Isolation

• “Concurrent transactions do not interfere with each other.”

• Each transaction is executed as if it was the only transaction

<table>
<thead>
<tr>
<th>T 1</th>
<th>T 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(X)</td>
<td>read(X)</td>
</tr>
<tr>
<td>read(Y)</td>
<td>read(Z)</td>
</tr>
<tr>
<td>write(X)</td>
<td>write(X)</td>
</tr>
<tr>
<td>write(Y)</td>
<td>write(Z)</td>
</tr>
<tr>
<td>commit</td>
<td>commit</td>
</tr>
</tbody>
</table>

• Initial example violates isolation property

Durability

• “The effects of a committed transaction are permanent.”

• If a server crashes and restarts, the effects of all committed transactions must be preserved

• Requires permanent storage (hard disks)
Conflicting operations

- What was the problem in the initial example?
  - Because of concurrency, T2 overwrote effect of T1
  - Isolation property was violated
- “Conflicting” operations

<table>
<thead>
<tr>
<th></th>
<th>read(X)</th>
<th>write(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 does</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read(X)</td>
<td>✓</td>
<td>Conflict</td>
</tr>
<tr>
<td>write(X)</td>
<td>Conflict</td>
<td>Conflict</td>
</tr>
</tbody>
</table>

- “Conflict” means things may go wrong
  - They don’t have to!
  - We cannot get rid of conflicts, but need to handle them
Conflicting operations, example

- **Two conflicts**
  - Conflict 1: write(X), read(X)
  - Conflict 2: write(Y), read(Y)

- **Conflicts exist regardless of ordering!**

<table>
<thead>
<tr>
<th>Interleaving 1</th>
<th>Interleaving 2</th>
<th>Interleaving 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(X)</td>
<td>read(X)</td>
<td>read(X)</td>
</tr>
<tr>
<td>read(Y)</td>
<td>read(Y)</td>
<td>read(Y)</td>
</tr>
<tr>
<td>write(X)</td>
<td>write(X)</td>
<td>write(X)</td>
</tr>
<tr>
<td>write(Y)</td>
<td>read(Y)</td>
<td>write(Y)</td>
</tr>
</tbody>
</table>

- Conflict 1: T2 before T1
- Conflict 2: T1 before T2

Conflict ordering

- **Conflicts exist regardless of ordering**
  - We cannot get rid of them!
- **But what does the ordering tell us?**

<table>
<thead>
<tr>
<th>read(X)</th>
<th>read(X)</th>
<th>read(X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(Y)</td>
<td>read(Y)</td>
<td>read(Y)</td>
</tr>
<tr>
<td>write(X)</td>
<td>write(X)</td>
<td>write(X)</td>
</tr>
<tr>
<td>write(Y)</td>
<td>read(Y)</td>
<td>write(Y)</td>
</tr>
</tbody>
</table>

- Conflict 1: T2 before T1
- Conflict 2: T1 before T2

Same as T2 T1!

- **Something weird is going on here!**

**Same as T1 T2!**
Serializability

- If the order on all conflicting operations is the same, ...
  - i.e., T1’s operation first for all conflicts
  - or T2’s operation first for all conflicts

- ... then the transactions are isolated!
  - they have the same effect as if the transactions were executed after one another!

- Definition: An interleaving is called serializable if all conflicting operations are executed in the same order

More examples

T1
- read(X)
- read(Y)
- write(X)
- write(Y)

T2
- read(X)
- read(Y)

- Two conflicts
  - Conflict 1: write(X), read(X)
  - Conflict 2: write(Y), read(Y)

Interleaving 4
- read(X)
- read(Y)
- write(X)
- write(Y)

Interleaving 5
- read(X)
- read(Y)
- write(X)
- write(Y)

Interleaving 6
- read(X)
- read(Y)
- write(X)
- write(Y)

✓ Serializable
✓ Serializable
X Not serializable
Back to the first example

\[ y_1 = \text{readBalance}(Y) \]
\[ g = \text{readBalance}(G) \]
\[ g = g - 1000 \]
\[ y_1 = y_1 + 1000 \]
\[ y_2 = y_2 + 500 \]
\[ \text{writeBalance}(Y, y_1) \]
\[ \text{writeBalance}(G, g) \]
\[ \text{writeBalance}(Y, y_2) \]
\[ \text{writeBalance}(U, u) \]

- Two conflicts
  - Conflict 1: read(Y), write(Y)
  - Conflict 2: read(Y), write(Y)

- Conflict 1: Grandma before uncle
- Conflict 2: Uncle before grandma

Non-solution: Global lock

- Idea: To execute a transaction, a client acquires a global lock

- Only two possible interleavings
  - read(A), read(B), write(A), write(B), read(A), read(B)
  - read(A), read(B), read(A), read(B), write(A), write(B)

- Lock - do not enter until T2 is finished.

- Lock - do not enter until T1 write are finished.

- Good: They are serializable!
- Very very bad: Global locks prevent all concurrency
Locks

• System-wide locking disallows concurrency
• More fine-grained locks per resource
  - Read lock: \texttt{rlock(X)}
  - Write lock: \texttt{wlock(X)}
  - Unlock: \texttt{unlock(X)} (for both lock types)
• If lock cannot be acquired, wait until it is available

<table>
<thead>
<tr>
<th>We want</th>
<th>No lock</th>
<th>Read lock</th>
<th>Write lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read lock</td>
<td>✓</td>
<td>✓</td>
<td>Wait</td>
</tr>
<tr>
<td>Write lock</td>
<td>✓</td>
<td>Wait</td>
<td>Wait</td>
</tr>
</tbody>
</table>

Another transaction has

2-phase locking

• Each transaction goes through two phases
  1. A growing phase, in which it may acquire locks
  2. A shrinking phase, in which it releases locks
• Always creates a serializable interleaving!
Example

- Execute T1, T2 using 2-phase locking

```
read(X) read(Y)
write(X) write(Y)
```

```
T1
```

```
T2
```

Deadlock

- What if two (or more) transactions wait for each other to release a lock?

```
rlock(X) read(X)
rlock(Y) read(Y)
wlock(Y) read(Y)
wait!
```

```
rlock(Y) read(Y)
```

```
wait!
wlock(X) read(Y)
```

```
wait!
wlock(Y) read(Y)
```

- Both transactions will wait forever!
Wait-for graph

- **Wait-for graph**
  - Each transaction is a node
  - Edge from T to U if T waits for a lock held by U
- Deadlock occurs if wait-for graph has a **cycle**

\[ T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1 \]

Deadlock!

\[ T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1 \]

No deadlock

- To resolve a deadlock, **abort** any one transaction in the cycle

Beyond 2-phase locking

- 2-phase locking ensures serializable interleavings
- Problems with 2-phase locking
  - Deadlocks can occur
  - Can be inefficient for write transactions given long read-only transaction
- Non-locking approaches
  - Optimistic concurrency control
Quiz 2

• Are the following statements true or false?

• If two transactions both execute write(X), then these two write operations are in conflict.
• An interleaving is called serializable if all pairs of conflicting operations are executed in the same order.
• Global locks are an efficient means of ensuring isolation.
• 2-phase locking ensures isolation.
• If two transactions are in a deadlock, only one of them will wait, while the other one continues.

Outline

• Motivation
• Transactions & ACID properties
• Isolation
  - Conflicts & serializability
  - 2-phase locking
  - Deadlocks

• Atomicity
  - 2-phase commit protocol
Atomicity in distributed systems

- Transaction involves multiple servers

Client \rightarrow \text{Coordinator} \rightarrow \begin{align*}
&\text{Server 1: } \text{write}(X) \\
&\text{Server 2: } \text{write}(Y)
\end{align*}

- Client accesses servers via a coordinator
- What about atomicity?

2-phase commit protocol

- Client asks coordinator to commit
- Must reach consensus on commit or abort
  - Even in the presence of failures!

- **2-phase commit protocol**
  - Protocol to ensure atomic distributed transactions
  - Phase 1: Voting
  - Phase 2: Completion

- (2-phase commit ≠ 2-phase locking)
Phase 1: voting

- Coordinator sends “can commit?” message to servers
- A server that cannot commit sends “no”
- A server that can commit sends “yes”
  - Before it sends “yes”, it must save all modifications to permanent storage

Phase 2: completion

- Coordinator collects votes
  (a) Failure or a “no” vote → coordinator sends “do abort” to all servers that have voted “yes”
  (b) All “yes” → coordinator sends “do commit” to all servers
- Servers handle “do abort” or “do commit”, respectively
Phase 2: completion

- Coordinator collects votes
  (a) Failure or a “no” vote → coordinator sends “do abort” to all servers that have voted “yes”
  (b) All “yes” → coordinator sends “do commit” to all servers
- Servers handle “do abort” or “do commit”, respectively
Failures

- A server crashes (and reboots) before voting
  - Server votes “no” after reboot
  - Or: coordinator times out waiting, sends “do abort” to all
- A server crashes (and reboots) after voting “no”
  - No problem
- A server crashes (and reboots) after voting “yes”
  - Server restores from permanent storage, waits for instruction from coordinator
  - Or: coordinator times out waiting, sends “do abort” to all
- The coordinator crashes after phase 1...

Server uncertainty

- Server that voted “yes” cannot abort
  - “Yes” means “I promise to commit when you ask me to.”

- **Server uncertainty**: time between a server’s “yes” vote and the coordinator’s “do commit” or “do abort” request
  - Server may not unlock resources!
Quiz 3

- Are the following statements true or false?

- The purpose of the 2-phase commit protocol is to ensure atomicity of distributed transactions.
- 2-phase commit is a variant of 2-phase locking.
- The 2-phase commit protocol is unable to cope with server failure.
- In phase 2, the coordinator sends a “do abort” message to all servers if it receives at least one “no” vote.

Reading

- Coulouris et al., Chapter 16.1 - 16.4
- Coulouris et al., Chapter 17.3, 2-phase commit protocol (without nested transactions)