Transactions and Concurrency Control

Distribuerade Informationssystem, 1DT060, HT 2013

Concurrency

concurrency *noun* \kən-ˈkə-rən(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.

```
y1=readBalance(Y)
g=readBalance(G)
g=g-1000
y1=y1+1000
writeBalance(Y,y1)
writeBalance(G,g)
```

Your uncle puts 500 SEK in your account.

```
y2=readBalance(Y)
u=readBalance(U)
u=u-500
y2=y2+500
writeBalance(Y,y2)
writeBalance(U,u)
```

Uncontrolled concurrency

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

```
y1=readBalance(Y)
g=readBalance(G)
                        y2=readBalance(Y)
                        u=readBalance(U)
g=g-1000
                        u=u-500
y1=y1+1000
                        y2=y2+500
writeBalance(Y,y1)
writeBalance(G,g)
                        writeBalance(Y,y2)
                        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**

Client

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


Server

ACID properties

- Desirable properties for transactions

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

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

Server

```
read(X)  ✓
read(Y)  ✓
write(X) ✓
~~write(Y)~~
~~commit~~
```

- 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

Client

withdraw 5 trillion SEK
from account G

abort

Server

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

```
T 1        T 2
read(X)
read(Y)
           read(X)
           read(Z)
write(X)
write(Y)
           write(X)
           write(Z)
commit
           commit
```

- 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)

Outline

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

Conflicting operations

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

T2 does

T1 does	read(X)	write(X)
read(X)	✓	Conflict
write(X)	Conflict	Conflict

- "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

T1	T2
read(X)	read(X)
read(Y)	read(Y)
write(X)	
write(Y)	

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

- Conflicts exist regardless of ordering!

Interleaving 1

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

Interleaving 2

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

Interleaving 3

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

Conflict ordering

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

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

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

Same as T2 T1!

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

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

???
Something weird is going on here!

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

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

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

✓ Serializable

Interleaving 5

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

✓ Serializable

Interleaving 6

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

✗ Not serializable

Back to the first example

```
y1=readBalance(Y)
g=readBalance(G)
                    y2=readBalance(Y)
                    u=readBalance(U)
g=g-1000
                    u=u-500
y1=y1+1000
                    y2=y2+500
writeBalance(Y,y1)
writeBalance(G,g)
                    writeBalance(Y,y2)
                    writeBalance(U,u)
```

Not serializable

- 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

Lock - do not enter until T2 is finished.

T1

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

T2

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

Lock - do not enter until T1 write are finished.

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

Locks

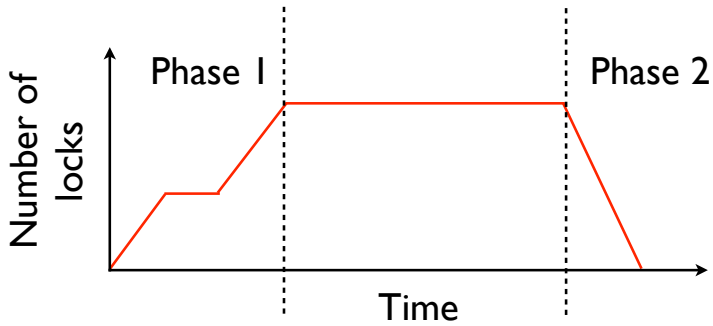
- System-wide locking disallows concurrency
- More fine-grained locks per resource
  - Read lock: `rlock(X)`
  - Write lock: `wlock(X)`
  - Unlock: `unlock(X)` (for both lock types)
- If *lock cannot be acquired*, wait until it is available

Another transaction has

We want	No lock	Read lock	Write lock
Read lock	✓	✓	Wait
Write lock	✓	Wait	Wait

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

T1	T2
read(X)	read(X)
read(Y)	read(Y)
write(X)	
write(Y)	

Time ↓

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

Time ↓

```
                . . .
   wait!        unlock(X)
                unlock(Y)
write(X)
wlock(Y)
write(Y)
unlock(X)
unlock(Y)
```

Deadlock

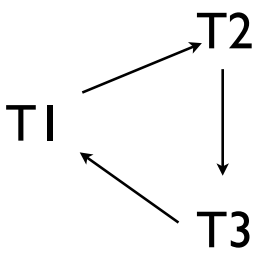
- What if two (or more) transactions wait for each other to release a lock?

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

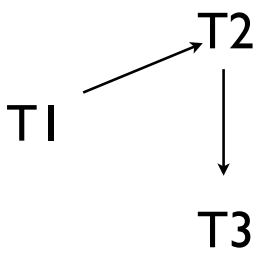
- Both transactions will wait forever!

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**



Deadlock!



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?

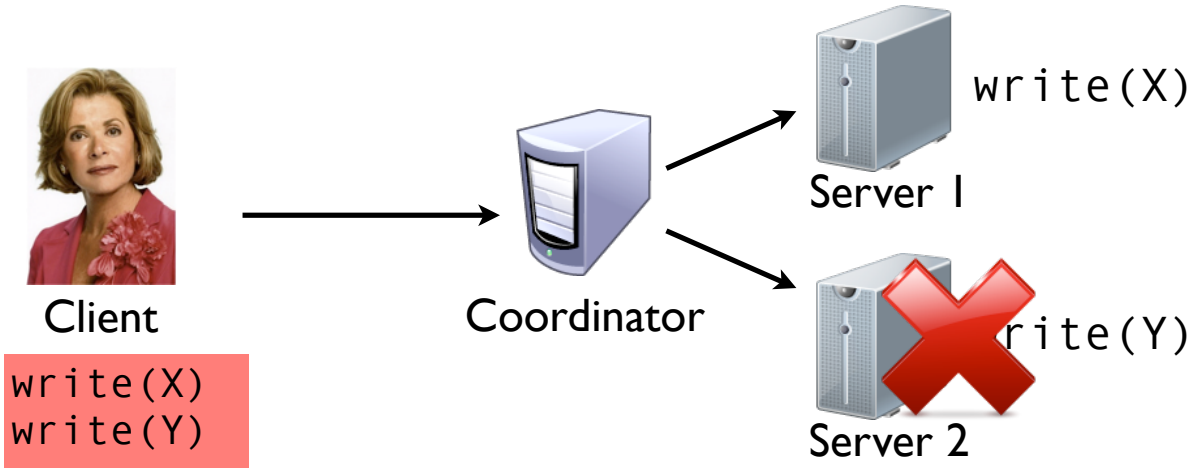
- If two transactions both execute write(X), then these two write operations are in conflict.
- An interleaving is called serializable if all pairs of conflicting operations are executed in the same order.
- Global locks are an efficient means of ensuring isolation.
- 2-phase locking ensures isolation.
- 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 accesses servers via a **coordinator**
- What about **atomicity**?

2-phase commit protocol

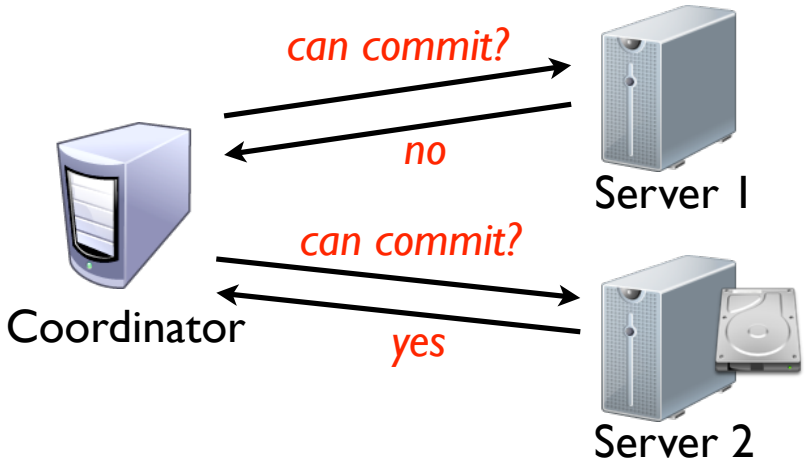
- Client asks coordinator to commit
- Must reach **consensus** on commit or abort
  - Even in the presence of failures!

- **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

Phase 2: completion

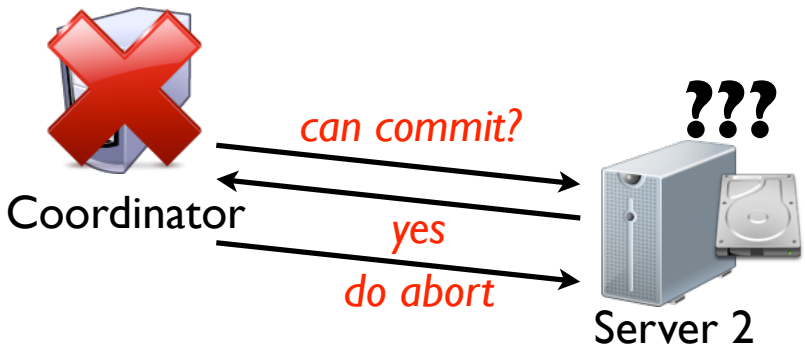
- Coordinator collects votes
  (a) Failure or a "*no*" vote → coordinator sends "*do abort*" to all servers that have voted "*yes*"
  (b) All "*yes*" → coordinator sends "*do commit*" to all servers
- Servers handle "*do abort*" or "*do commit*", respectively

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)