

# Distributed Systems

CS425/ECE428

*Acknowledgements for the materials: Indy Gupta and Radhika Mittal*

# Agenda for the next 2-3 classes

- Transaction Processing and Concurrency Control
  - Chapter 16
    - Transaction semantics: ACID
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering
- First focus on transactions executed on a single server.
- Look into distributed transactions later (Chapter 17)

# Today's Agenda

- Transaction Processing and Concurrency Control
  - Chapter 16
    - Transaction semantics: ACID
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering
- First focus on transactions executed on a single server.
- Look into distributed transactions later (Chapter 17)

# Transaction

- Series of *operations* executed by a client on a server (or a set of servers).
- Example: Switch from T4 to T3 section:  
`rosters.remove("ece428", "t4", student.name)`  
`student.schedule.remove("ece428", "t4")`  
`student.schedule.add("ece428", "t3")`  
`rosters.add("ece428", "t3", student.name)`

# Transaction

- Another example:

Client code:

```
int transaction_id = openTransaction();
x = server.getFlightAvailability(ABC123, date);           // read(ABC123, date)
if (x > 0)
    y = server.bookTicket(ABC123, date);                 // write(ABC123, date)
server.putSeat(y, "aisle");                               // write(ABC123, date)
closeTransaction(transaction_id);
```

# Transaction Properties

- Atomic: all-or-nothing
  - Transaction either executes completely or not at all
- Consistent: required rules are maintained
- Isolation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- Durability: values preserved even after crashes

# Transaction Properties

- **A**tomic: all-or-nothing
  - Transaction either executes completely or not at all
- **C**onsistent: rules maintained
- **I**solation: multiple transactions do not interfere with each other
  - Equivalent to running transactions in isolation
- **D**urability: values preserved even after crashes

**ACID properties**

# Atomicity

- All-or-nothing
  - Transaction either executes completely or not at all
- What can happen after partial execution?

```
rosters.remove("ece428", "t4", student.name)
student.schedule.remove("ece428", "t4")
student.schedule.add("ece428", "t3")
rosters.add("ece428", "t3", student.name)
```



# Atomicity

- All-or-nothing
  - Transaction either executes completely or not at all
- Make tentative updates to data.
- **Commit** transaction to make tentative updates permanent.
- **Abort** transaction to roll back to previous values.

# Consistency

Various rules about state of objects must be maintained:

- E.g. class enrollment limit, schedule can't conflict
- Account balances have to stay positive
- Consistency must be maintained at end of transaction.
- Checked at commit time, abort if not satisfied

```
rosters.remove("ece428", "t4", student.name)  
student.schedule.remove("ece428", "t4")  
student.schedule.add("ece428", "t3")  
rosters.add("ece428", "t3", student.name)
```

# Durability

- Committed transactions must persist:
  - Client crashes
  - Server crashes
- How do we ensure this?
  - Replication
  - Permanent storage

# Isolation

Multiple clients may execute transactions concurrently on one server.

What could go wrong?

# What could go wrong?

## Transaction T1

`x = getSeats(ABC123);`

`// x = 10`

`if(x > 1)`

`x = x - 1;`

`write(x, ABC123);`

`commit`

## Transaction T2

`x = getSeats(ABC123);`

`if(x > 1)                   // x = 10`

`x = x - 1;`

`write(x, ABC123);`

`commit`

At Server: seats = 10

seats = 9

seats = 9

# I. Lost Update Problem

## Transaction T1

`x = getSeats(ABC123);`

`// x = 10`

`if(x > 1)`

`x = x - 1;`

`write(x, ABC123);`

`commit`

## Transaction T2

`x = getSeats(ABC123);`

`if(x > 1)`

`// x = 10`

`x = x - 1;`

`write(x, ABC123);`

`commit`

At Server: seats = 10

seats = 9

seats = 9

**T1's or T2's update was lost!**

# What else could go wrong?

## Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);
```

**// ABC123 = 5 now**

```
write(y+5, ABC789);
```

```
commit
```

## Transaction T2

```
x = getSeats(ABC123);  
y = getSeats(ABC789);
```

**// x = 5, y = 15**

```
print("Total:" x+y);
```

**// Prints "Total: 20"**

```
commit
```

At Server:

ABC123 = 10

ABC789 = 15

## 2. Inconsistent Retrieval Problem

### Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);
```

**// ABC123 = 5 now**

```
write(y+5, ABC789);
```

```
commit
```

### Transaction T2

```
x = getSeats(ABC123);  
y = getSeats(ABC789);
```

**// x = 5, y = 15**

```
print("Total:" x+y);
```

**// Prints "Total: 20"**

```
commit
```

At Server:

ABC123 = 10

ABC789 = 15

**T2's sum is the  
wrong value!  
Should have been  
"Total: 25"**



# Isolation

Multiple clients executing transactions concurrently on one server.

What could go wrong?

- Lost Update Problem
- Inconsistent Retrieval Problem

*How to prevent transactions from affecting each other?*

# Isolation

*How to prevent transactions from affecting each other?*

- Option 1: Execute them serially at the server (one at a time).
  - Grab a global lock before executing any transaction, release the lock after the transaction has committed (or aborted).
- But this reduces number of concurrent transactions
  - *Transactions per second* directly related to revenue of companies
    - This metric needs to be maximized

*Goal: increase concurrency while maintaining correctness (ACID).*

# Concurrent Transactions

*Goal: increase concurrency while maintaining correctness (ACID).*

- How do we increase concurrency?
  - Instead of targeting strict serial execution, target **serial equivalence**.

# Interleaving

- An ordered sequence of the operations across multiple transactions, where each transaction's operations follows the order defined by the transaction.
- E.g., if  $T_1 = \{op_1, op_2, op_3\}$  and  $T_2 = \{op^4, op^5, op^6\}$  then  $O = \{op_1, op_2, op^4, op_3, op^5, op^6\}$  is a sample interleaving.

# Interleaving: Another example

## Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);
```

```
write(y+5, ABC789);
```

```
commit
```

## Transaction T2

```
x = getSeats(ABC123);  
y = getSeats(ABC789);
```

```
print("Total:" x+y);
```

```
commit
```

## Interleaving

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);  
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(y+5, ABC789);
```

```
print("Total:" x+y);
```

```
commit
```

```
commit
```

# Concurrent Transactions

- Allowing transaction operations to be interleaved with one-another increases concurrency.
- To avoid transactions from affecting one another, the interleaving of operations across transactions must be *serially equivalent*.

# Serial Equivalence

- An interleaving (say  $O$ ) of transaction operations is serially equivalent iff (if and only if):
  - There is some ordering ( $O'$ ) of those transactions, one at a time,
  - where the operations of each transaction occur consecutively (in a batch)
  - which gives the same end-result (for all objects and transactions) as the original interleaving  $O$
- Says: Cannot distinguish end-result of real operation  $O$  from (fake) serial transaction order  $O'$
- E.g., if  $T_1 = \{op_1, op_2, op_3\}$  and  $T_2 = \{op^4, op^5, op^6\}$   
then  $O = \{op_1, op_2, op^4, op_3, op^5, op^6\}$  is a serially equivalent, if:
  - end result of  $O$  is same as  $\{op_1, op_2, op_3, op^4, op^5, op^6\}$
  - **Or** end result of  $O$  is same as  $\{op^4, op^5, op^6, op_1, op_2, op_3, \}$

# I. Lost Update Problem

## Transaction T1

`x = getSeats(ABC123);`

`// x = 10`

`if(x > 1)`

`x = x - 1;`

`write(x, ABC123);`

`commit`

## Transaction T2

`x = getSeats(ABC123);`

`if(x > 1)`

`// x = 10`

`x = x - 1;`

`write(x, ABC123);`

`commit`

At Server: seats = 10

seats = 9

seats = 9

**T1's or T2's update was lost!**  
**Not serially equivalent.**



## 2. Inconsistent Retrieval Problem

### Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);
```

**// ABC123 = 5 now**

```
write(y+5, ABC789);
```

```
commit
```

### Transaction T2

```
x = getSeats(ABC123);  
y = getSeats(ABC789);
```

**// x = 5, y = 15**

```
print("Total:" x+y);
```

**// Prints "Total: 20"**

```
commit
```

At Server:

ABC123 = 10

ABC789 = 15

**T2's sum is the  
wrong value!  
Should have been  
"Total: 25"**

**Not serially equivalent.**

# Checking for Serial Equivalence

- An operation has an **effect** on
  - The server object if it is a write
  - The client (returned value) if it is a read
- Two operations are said to be conflicting operations, if their **combined effect** depends on the **order** they are executed
  - read(x) and write(x): conflicting
  - write(x) and read(x): conflicting
  - write(x) and write(x): conflicting
  - read(x) and read(x): NOT conflicting
    - swapping them doesn't change their effects
  - read/write(x) and read/write(y): NOT conflicting
    - ok to swap them as they access different objects.

# Checking for Serial Equivalence (cont.)

- *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.*
  - Take all pairs of conflict operations, one from T1 and one from T2
  - If the T1 operation was reflected first on the server, mark the pair as “(T1,T2)”, otherwise mark it as “(T2,T1)”
  - All pairs should be marked as either “(T1,T2)” or all pairs should be marked as “(T2,T1)”.