# Distributed Transactions and Concurrency

CS425/ECE 428

Nikita Borisov

# Topics for Today

- Transaction semantics: ACID
- Isolation and serial equivalence
- Conflicting operations
- Two-phase locking

# Example transaction

Switch from T3 to TU4 section

rosters.remove(“ece428”, “t3”, student.name)
student.schedule.remove(“ece428”, “t3”)
student.schedule.add(“ece428”, “tu4”)
rosters.add(“ece428”, “tu4”, student.name)

# Transaction Properties

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

# Atomicity

What can happen after partial execution?

rosters.remove(“ece428”, “t3”, student.name)
student.schedule.remove(“ece428”, “t3”)
student.schedule.add(“ece428”, “tu4”)
rosters.add(“ece428”, “tu4”, student.name)

# Atomicity

- Make *tentative* updates to data
- *Commit* transaction to make tentative updates permanent
- *Abort* transaction to roll back to previous values
- How to do this in a distributed system?
  - Will discuss next week.

# Consistency

Various *rules* about state of objects must be maintained

Examples?

- 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", "t3", student.name)

student.schedule.remove("ece428", "t3")

student.schedule.add("ece428", "tu4")

rosters.add("ece428", "tu4", student.name)

# Durability

Committed transactions must persist

- Client crashes
- Server crashes

How do we ensure this?

- Replication
- Permanent storage

# Isolation

T1: add 1% dividend to account A

```
x := A.getBalance()

A.setBalance(x * 1.01)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()

A.setBalance(t-100)
B.setBalance(u+100)
```

Lost Update Problem

# Isolation

T1: add 1% dividend to C based on balances of A and B

```
x := A.getBalance()
y := B.getBalance()
z := C.getBalance()

C.setBalance((x+y)*0.01+z)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()
A.setBalance(t-100)

B.setBalance(u+100)
```

Inconsitent Retrieval Problem

# Serial Equivalence

Effect of two transactions should be equivalent to running one tx to completion, then running other.

T1: add 1% dividend to C based on balances of A and B

```
x := A.getBalance()
y := B.getBalance()
z := C.getBalance()

C.setBalance((x+y)*0.01+z)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()
A.setBalance(t-100)
B.setBalance(u+100)
```

# Serial Equivalence

Effect of two transactions should be equivalent to running one tx to completion, then running other.

T1: add 1% dividend to account A

```
x := A.getBalance()
A.setBalance(x * 1.01)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()

A.setBalance(t-100)
B.setBalance(u+100)
```

# Achieving Serial Equivalence

How do we achieve serial equivalence?

Option 1: Serialize all transactions

- Grab a (global) lock on (all?) accounts at start of transaction
- Release at commit / abort time

Can we do better?

# 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 X / write X
  - E.g., write X / write X
  - E.g., write Y / write Z
  - E.g., read X / read X

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.

# Conflicting Operations

What are all the conflicts?

T1: add 1% dividend to C based on balances of A and B

```
x := A.getBalance()
y := B.getBalance()
z := C.getBalance()

C.setBalance((x+y)*0.01+z)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()
A.setBalance(t-100)
B.setBalance(u+100)
```

# Conflicting Operations

What are all the conflicts?

T1: add 1% dividend to account A

```
x := A.getBalance()
A.setBalance(x * 1.01)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()

A.setBalance(t-100)
B.setBalance(u+100)
```

# Conflict Ordering

T1: add 1% dividend to account A

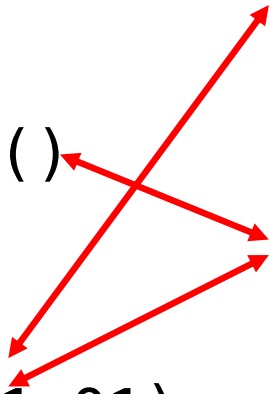
```
x := A.getBalance()

A.setBalance(x * 1.01)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()

A.setBalance(t-100)
B.setBalance(u+100)
```



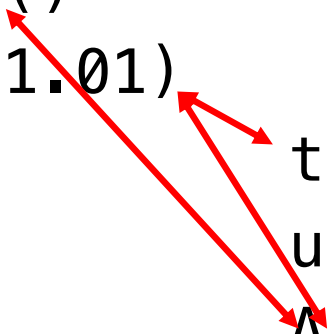
# Serially Equivalent

T1: add 1% dividend to account A

```
x := A.getBalance()
A.setBalance(x * 1.01)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()
A.setBalance(t-100)
B.setBalance(u+100)
```



# Serially Equivalent

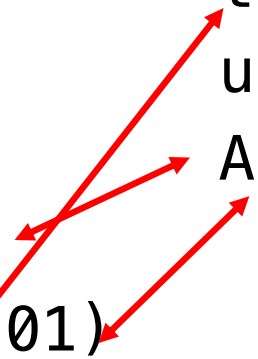
T1: add 1% dividend to account A

T2: transfer 100 from A to B

```
x := A.getBalance()
A.setBalance(x * 1.01)
```

```
t := A.getBalance()
u := B.getBalance()
A.setBalance(t-100)
```

```
B.setBalance(u+100)
```



# Conflict Ordering

T1: add 1% dividend to C based on balances of A and B

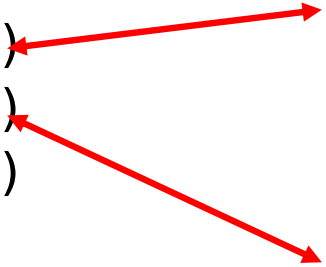
```
x := A.getBalance()
y := B.getBalance()
z := C.getBalance()

C.setBalance((x+y)*0.01+z)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()
A.setBalance(t-100)

B.setBalance(u+100)
```



# Serially equivalent

T1: add 1% dividend to C based on balances of A and B

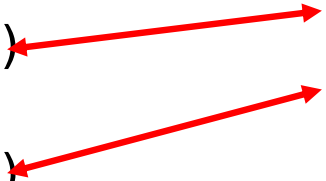
```
x := A.getBalance()

y := B.getBalance()
z := C.getBalance()

C.setBalance((x+y)*0.01+z)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()
A.setBalance(t-100)
B.setBalance(u+100)
```



# Serially equivalent

T1: add 1% dividend to C based on balances of A and B

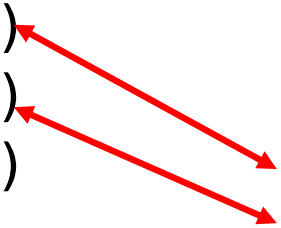
```
x := A.getBalance()
y := B.getBalance()
z := C.getBalance()

C.setBalance((x+y)*0.01+z
)
```

T2: transfer 100 from A to B

```
t := A.getBalance()
u := B.getBalance()

A.setBalance(t-100)
B.setBalance(u+100)
```



# Locking

T1: add 1% dividend to C based on balances of A and B

```
Lock A, B, C
x := A.getBalance()
y := B.getBalance()
z := C.getBalance()
C.setBalance((x+y)*0.01+z)
Unlock A, B, C
```

T2: transfer 100 from A to B

```
Lock A
t := A.getBalance()
A.setBalance(t-100)
Unlock A
```

```
Lock B
u := B.getBalance()
B.setBalance(u+100)
Unlock B
```

# Two-phase Locking

Locks are acquired before accessing objects

Locks are kept until transaction commits / aborts

Two phases:

- Growing phase: new locks acquired, no locks released
- Shrinking phase: all locks released at once

# Two-Phase Locking

T1: add 1% dividend to C based on balances of A and B

*Try to lock A, wait*

*Lock A*
x := A.getBalance()
*Lock B*
y := B.getBalance()
*Lock C*
z := C.getBalance()
C.setBalance((x+y)∗0.01+z)
*Unlock A, B, C*

T2: transfer 100 from A to B

*Lock A*
t := A.getBalance()
A.setBalance(t−100)

*Lock B*
u := B.getBalance()
B.setBalance(u+100)
*Unlock A, B*

# Two-phase locking correctness

Consider two transactions T1, T2

Let

- t1a be time that T1 acquires its last lock
- t1r be the time that T1 releases its first lock
- Likewise t2a, t2r

Claim 1: if T1, T2 have any conflicts, then t1r < t2a or t1r < t2a

Claim 2: if t1r < t2a then all conflicts must be in order T1 -> T2