Transactions

SWE 622, Spring 2017
Distributed Software Engineering
Review: HW2
Sequential Consistency: Quiz

- There is some *total order* of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)
Sequential Consistency: Quiz

- There is some total order of operations so that:
- Each CPUs operations appear in order
- All CPUs see results according to that order (read most recent writes)

<table>
<thead>
<tr>
<th></th>
<th>W(X) a</th>
<th>W(X) b</th>
<th>R(X) b</th>
<th>R(X) a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Consistency: Quiz

- There is some *total order* of operations so that:
- Each CPUs operations appear in order
- All CPUs see results according to that order (read most recent writes)

Sequentially consistent but not strictly consistent.

\[ W(X)b, R(X)b, R(X)b, W(X)a, R(X)a, R(X)a \]
Sequential Consistency: Quiz

- There is some *total order* of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)
Sequential Consistency: Quiz

- There is some *total order* of operations so that:
  - Each CPUs operations appear in order
  - All CPUs see results according to that order (read most recent writes)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W(X) a</td>
<td>W(X) b</td>
<td>R(X) b</td>
<td>R(X) a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(X) a</td>
<td>R(X) b</td>
</tr>
</tbody>
</table>
Sequential Consistency: Quiz

- There is some *total order* of operations so that:
- Each CPUs operations appear in order
- All CPUs see results according to that order (read most recent writes)

<table>
<thead>
<tr>
<th>P1</th>
<th>W(X) a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>W(X) b</td>
</tr>
<tr>
<td>P3</td>
<td>R(X) b</td>
</tr>
<tr>
<td>P4</td>
<td>R(X) a</td>
</tr>
</tbody>
</table>

*Not sequentially consistent*
Causal Consistency
Causal Consistency

<table>
<thead>
<tr>
<th></th>
<th>W(X)a</th>
<th></th>
<th>W(X)c</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>R(X)a</td>
<td>W(X)b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>R(X)a</td>
<td></td>
<td>R(X)c</td>
<td>R(X)b</td>
</tr>
<tr>
<td>P4</td>
<td>R(X)a</td>
<td></td>
<td>R(X)b</td>
<td>R(x)c</td>
</tr>
</tbody>
</table>
Causal Consistency

Causally Consistent. \( W(X)b \) and \( W(X)c \) are not related, hence could have happened one either order.

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th></th>
<th>P2</th>
<th></th>
<th>P3</th>
<th></th>
<th>P4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W(X)a</td>
<td></td>
<td>R(X)a</td>
<td>W(X)b</td>
<td></td>
<td>R(X)a</td>
<td>R(X)c</td>
<td>R(X)b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R(X)a</td>
<td></td>
<td></td>
<td></td>
<td>R(X)b</td>
<td></td>
</tr>
</tbody>
</table>

J. Bell

GMU SWE 622 Spring 2017
Causal Consistency
# Causal Consistency

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>W(X)$_a$</td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>R(X)$_a$</td>
<td>W(X)$_b$</td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td>R(x)$_b$</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td>R(x)$_a$</td>
</tr>
</tbody>
</table>
Causal Consistency

<table>
<thead>
<tr>
<th>P1</th>
<th>W(X)a</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>R(X)a</td>
</tr>
<tr>
<td>P3</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td></td>
</tr>
</tbody>
</table>

**NOT Causally Consistent.** X couldn’t have been b after it was a.
Causal Consistency

**NOT Causally Consistent.** X couldn’t have been b after it was a

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W(X)a</td>
<td>R(X)a</td>
<td>R(x)b</td>
<td>R(x)a</td>
</tr>
<tr>
<td></td>
<td>W(X)b</td>
<td>W(X)b</td>
<td>R(x)b</td>
<td>R(x)b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W(X)a</td>
<td>W(X)b</td>
<td>R(x)b</td>
<td>R(x)a</td>
</tr>
<tr>
<td></td>
<td>W(X)b</td>
<td>W(X)b</td>
<td>R(x)b</td>
<td>R(x)b</td>
</tr>
</tbody>
</table>
Causal Consistency

<table>
<thead>
<tr>
<th></th>
<th>W(X)a</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td>R(X)a</td>
<td>W(X)b</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td>R(x)b</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td>R(x)a</td>
</tr>
</tbody>
</table>

NOT Causally Consistent. X couldn’t have been b after it was a

<table>
<thead>
<tr>
<th></th>
<th>W(X)a</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td>W(X)b</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td>R(x)b</td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td>R(x)a</td>
</tr>
</tbody>
</table>

Causally Consistent. X can be a or b concurrently
Eventual Consistency

• Allow stale reads, but ensure that reads will **eventually** reflect the previously written values
  • Eventually: milliseconds, seconds, minutes, hours, years…

• Writes are NOT ordered as executed
  • Allows for conflicts. Consider: Dropbox

• Git is eventually consistent
Eventual Consistency

• More concurrency than strict, sequential or causal
  • These require **highly available** connections to send messages, and generate lots of chatter

• Far looser requirements on network connections
  • Partitions: OK!
  • Disconnected clients: OK!
  • Always available!

• Possibility for conflicting writes :(
Redis Replication

Client

Master

Slave

Slave

Client
Redis Replication

1: Write key “foo” = “bar”
Redis Replication

1: Write key “foo” = “bar”

2: Acknowledge write
Redis Replication

1: Write key “foo” = “bar”

2: Acknowledge write

3: Send update
Redis Replication

1: Write key "foo" = "bar"

2: Acknowledge write

3: Send update
Redis Replication

1: Write key “foo” = “bar”

2: Acknowledge write

3: Send update

Client

Master

Slave

Slave

Read foo

foo=???

foo=bar

foo=bar

foo=bar
Redis “Wait” command

Client

Master

Slave

Slave
Redis “Wait” command

1: Write key “foo” = “bar”. WAIT
Redis “Wait” command

1: Write key “foo” = “bar”. WAIT

2: Send update
Redis “Wait” command

1: Write key “foo” = “bar”. WAIT

2: Send update

Client

Master

Slave

Slave

Client
Redis “Wait” command

1: Write key “foo” = “bar”. WAIT

2: Send update

3: Acknowledge

foo=bar
Redis “Wait” command

1: Write key “foo” = “bar”. WAIT

4: Acknowledge write

2: Send update

3: Acknowledge

foo=bar
Redis “Wait” command

1: Write key “foo” = “bar”. WAIT

4: Acknowledge write

2: Send update

3: Acknowledge

1. Client

2. Master

3. Slave

4. Slave

Client

Master

Slave

Slave

foo=bar

Write key “foo” = “bar”. WAIT

Acknowledge write

Send update

foo=bar

foo=bar

foo=???

Read foo

foo=???

Read foo
HW3: Replicate Redis

• What happens if Redis fails?
• Solution:
  • Redis has built in replication!
• What consistency guarantees does that provide?
• We want to maintain what we’ve got.
• You’ll use WAIT after writes
• All writes -> master, reads -> slave (note: now each client has its own redis slave)
• Add heartbeat to know how many replicas there are
Today

• Transactions

• 2 Phase Locking

• 2 Phase Commit

• Write Ahead Logging
Transactions

• The past few weeks we’ve talked about consistency of individual reads/writes

• How can we provide some consistency guarantees across operations

• Transaction: unit of work (grouping) of operations
  • Begin transaction
  • Do stuff
  • Commit OR abort
Properties of Transactions
Properties of Transactions

• Traditional properties: ACID
Properties of Transactions

• Traditional properties: ACID

• **Atomicity**: transactions are “all or nothing”
Properties of Transactions

• Traditional properties: ACID

- **Atomicity**: transactions are “all or nothing”

- **Consistency**: Guarantee some basic properties of data; each transaction leaves the database in a valid state
Properties of Transactions

• Traditional properties: ACID

• **Atomicity**: transactions are “all or nothing”

• **Consistency**: Guarantee some basic properties of data; each transaction leaves the database in a valid state

• **Isolation**: Each transaction runs as if it is the only one; there is some valid serial ordering that represents what happens when transactions run concurrently
Properties of Transactions

- **Traditional properties: ACID**
  - **Atomicity**: transactions are “all or nothing”
  - **Consistency**: Guarantee some basic properties of data; each transaction leaves the database in a valid state
  - **Isolation**: Each transaction runs as if it is the only one; there is some valid serial ordering that represents what happens when transactions run concurrently
  - **Durability**: Once committed, updates cannot be lost despite failures
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

What can go wrong here?
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount){
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

`transferMoney(P1, P2, 100)`  `transferMoney(P1, P2, 200)`
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount){
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P1.balance = 200 - 200 = 0</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    if (from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

```plaintext
<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P1.balance = 200 - 200 = 0</td>
</tr>
<tr>
<td>P2.balance = 200 + 100 = 300</td>
<td></td>
</tr>
</tbody>
</table>
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td><strong>P1.balance = 200 - 200 = 0</strong></td>
</tr>
<tr>
<td>P2.balance = 200 + 100 = 300</td>
<td>P2.balance = 300 + 200 = 500</td>
</tr>
<tr>
<td>return true;</td>
<td></td>
</tr>
</tbody>
</table>
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

```
transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0
P2.balance = 200 + 100 = 300
return true;
```

```
transferMoney(P1, P2, 200)
P2.balance (200) > 200
P1.balance = 200 - 200 = 0
P2.balance = 300 + 200 = 500
return true;
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount){
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td><strong>P1.balance = 200 - 200 = 0</strong></td>
</tr>
<tr>
<td>P2.balance = 200 + 100 = 300</td>
<td>P2.balance = 300 + 200 = 500</td>
</tr>
<tr>
<td>return true;</td>
<td>return true;</td>
</tr>
</tbody>
</table>

OK, we know how to solve this one (we need sequential consistency on P1.balance, which we’d get normally given the way the code is written above if balance is volatile).
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

`transferMoney(P1, P2, 100)`  `transferMoney(P1, P2, 200)`
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

```
transferMoney(P1, P2, 100)
P1.balance (200) >= 100

transferMoney(P1, P2, 200)
P2.balance (200) > 200
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    if (from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td></td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P2.balance = 200 + 100 = 300</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P2.balance = 200 + 100 = 300</td>
</tr>
<tr>
<td>P2.balance = 200 + 100 = 300</td>
<td>P1.balance = 100 - 200 = -100</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    if (from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P2.balance (200) &gt; 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P1.balance = 100 - 200 = -100</td>
</tr>
<tr>
<td>P2.balance = 200 + 100 = 300</td>
<td>P2.balance = 300 + 200 = 500</td>
</tr>
</tbody>
</table>

return true;
boolean transferMoney(Person from, Person to, float amount) {
    if (from.balance >= amount) {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}

transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0
P2.balance = 200 + 100 = 300
return true;

transferMoney(P1, P2, 200)
P2.balance (200) > 200
P1.balance = 100 - 200 = -100
P2.balance = 300 + 200 = 500
return true;
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.balance = to.balance + amount;
        return true;
    }
    return false;
}
```

What's wrong here?
Need isolation (prevent overdrawing)

```
transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0
P2.balance = 200 + 100 = 300
return true;

transferMoney(P1, P2, 200)
P2.balance (200) > 200
P1.balance = 100 - 200 = -100
P2.balance = 300 + 200 = 500
return true;
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

`transferMoney(P1, P2, 100)`  `transferMoney(P1, P2, 200)`
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td></td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
    }
    return false;
}
```

transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0

transferMoney(P1, P2, 200)
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P1.balance = 200 - 100 = 0</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P2.balance = 200 + 100 = 300</td>
</tr>
</tbody>
</table>
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
    return false;
}
}

transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0
P2.balance = 200 + 100 = 300
return true;

transferMoney(P1, P2, 200)
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
            {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

- **transferMoney(P1, P2, 100)**
  - P1.balance (200) >= 100
  - P1.balance = 200 - 100 = 0
  - P2.balance = 200 + 100 = 300
  - return true;

- **transferMoney(P1, P2, 200)**
  - P1.balance <= 200
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

**transferMoney(P1, P2, 100)**
- P1.balance (200) >= 100
- P1.balance = 200 - 100 = 0
- P2.balance = 200 + 100 = 300
- return true;

**transferMoney(P1, P2, 200)**
- P1.balance <= 200
- return false;
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
}
```

Adding a lock: prevents accounts from being overdrawn

**transferMoney(P1, P2, 100)**
- P1.balance (200) >= 100
- P1.balance = 200 - 100 = 0
- P2.balance = 200 + 100 = 300
- return true;

**transferMoney(P1, P2, 200)**
- P1.balance <= 200
- return false;
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}

Adding a lock: prevents accounts from being overdrawn

transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0
P2.balance = 200 + 100 = 300
return true;

transferMoney(P1, P2, 200)
P1.balance <= 200
return false;

But: shouldn’t we lock on to also?
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

`transferMoney(P1, P2, 100)`  `transferMoney(P1, P2, 200)`
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized(from, to) {
        if(from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

transferMoney(P1, P2, 100)  transferMoney(P1, P2, 200)
P1.balance (200) >= 100
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized(from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td></td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td></td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }  
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P1.balance = 200 - 100 = 0</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P2.balance = 200 + 100 = 300</td>
</tr>
</tbody>
</table>

P1.balance (200) >= 100
P1.balance = 200 - 100 = 0
P2.balance = 200 + 100 = 300
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P1.balance (200) &gt;= 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>P1.balance = 200 + 100 = 300</td>
</tr>
<tr>
<td>return true;</td>
<td>return false;</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

transferMoney(P1, P2, 100)
- P1.balance (200) >= 100
- P1.balance = 200 - 100 = 0
- P2.balance = 200 + 100 = 300
- return true;

transferMoney(P1, P2, 200)
- P1.balance <= 200
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P1.balance &lt;= 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>return false</td>
</tr>
<tr>
<td>P2.balance = 200 + 100 = 300</td>
<td>P1.balance = 200</td>
</tr>
<tr>
<td>return true;</td>
<td>return false</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P1.balance &lt;= 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td></td>
</tr>
<tr>
<td>P2.balance = 200 + 100 = 300</td>
<td></td>
</tr>
<tr>
<td>return true;</td>
<td>return false;</td>
</tr>
</tbody>
</table>

Locking on both from, to at same time
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

transferMoney(P1, P2, 100)  transferMoney(P1, P2, 200)
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
            {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P1.balance (200) &gt;= 200</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
    return false;
    }
}
```

transferMoney(P1, P2, 100)
P1.balance (200) >= 100
P1.balance = 200 - 100 = 0

transferMoney(P1, P2, 200)
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized (from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

```text
<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td></td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td></td>
</tr>
</tbody>
</table>
```

P1.balance (200) >= 100
P1.balance = 200 - 100 = 0
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount){
    synchronized(from, to){
        if(from.balance >= amount)
        {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

- `transferMoney(P1, P2, 100)`
  - P1.balance (200) >= 100
  - P1.balance = 200 - 100 = 0

- `transferMoney(P1, P2, 200)`
  - P1.balance <= 200
### Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized(from, to) {
        if(from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
    }
    return false;
}
```

<table>
<thead>
<tr>
<th>transferMoney(P1, P2, 100)</th>
<th>transferMoney(P1, P2, 200)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1.balance (200) &gt;= 100</td>
<td>P1.balance &lt;= 200</td>
</tr>
<tr>
<td>P1.balance = 200 - 100 = 0</td>
<td>return false;</td>
</tr>
</tbody>
</table>
Transactions: Classic Example

```java
boolean transferMoney(Person from, Person to, float amount) {
    synchronized(from, to) {
        if (from.balance >= amount) {
            from.balance = from.balance - amount;
            to.balance = to.balance + amount;
            return true;
        }
        return false;
    }
    return false;
}
```

Problem: P1.balance was deducted P2.balance not incremented! ("Atomicity violation")
2-phase locking

- Simple solution for isolation
- Phase 1: acquire locks (all that you might need)
- Phase 2: release locks
  - You can’t get any more locks after you release any
  - Typically: locks released when you say “commit” or “abort”
NOT 2-phase locking

```java
boolean transferMoney(Person from, Person to, float amount){
    from.lock();
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        from.unlock();
        to.lock();
        to.balance = to.balance + amount;
        to.unlock();
        return true;
    }
    else
    {
        from.unlock();
        return false;
    }
}
```
NOT 2-phase locking

```java
boolean transferMoney(Person from, Person to, float amount) {
    from.lock();
    if (from.balance >= amount) {
        from.balance = from.balance - amount;
        from.unlock();
        to.lock();
        to.balance = to.balance + amount;
        to.unlock();
        return true;
    } else {
        from.unlock();
        return false;
    }
}
```

Invalid: other transactions could read an inconsistent system state at this point!
2-phase locking

```java
boolean transferMoney(Person from, Person to, float amount){
    from.lock();
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.lock();
        to.balance = to.balance + amount;
        to.unlock();
        from.unlock();
        return true;
    }
    else
        from.unlock();
    return false;
}
```
2-phase locking

```java
boolean transferMoney(Person from, Person to, float amount){
    from.lock();
    if(from.balance >= amount)
    {
        from.balance = from.balance - amount;
        to.lock();
        to.balance = to.balance + amount;
        to.unlock();
        from.unlock();
        return true;
    }
    else
    {
        from.unlock();
        return false;
    }
}
```

Might deadlock if one transaction gives from P1->P2, other P2->P1
Avoiding Deadlocks

- Remember: dining philosophers
- Easiest fix: always acquire locks in the same order
  - E.g., first acquire lock on person with older account, then other person?
- Also: get all locks at same time
  - Not really practical
- Alternatively: timeouts
Distributing Transactions

• System model: data stored in multiple locations, multiple servers participating in a single transaction. One server pre-designated “coordinator”

• Failure model: messages can be delayed or lost, servers might crash, but have persistent storage to recover from
Distributed Transactions

- Coordinator: Begins a transaction
  - Assigns a unique transaction ID
  - Responsible for commit + abort
  - In principle, any client can be the coordinator, but all participants need to agree on who is the coordinator
- Participants: everyone else who has the data used in the transaction
1-Phase Transaction Commit

• Naive protocol: coordinator broadcasts out “commit!” continuously until participants all say “OK!”

• Problem: what happens when a participants fails during commit? How do the other participants know that they shouldn’t have really committed and they need to abort?
1-Phase Commit

Commit!

Commit!

Commit!

Commit!

Commit!

Commit!
1-Phase Commit

Commit!

OK! OK! Nope!

J. Bell
GMU SWE 622 Spring 2017
1-Phase Commit

We couldn’t successfully commit on all 3 machines. But 1-phase commit has no way to go back!
2-Phase Commit
2-Phase Commit

• Separate the commit into two steps:
2-Phase Commit

• Separate the commit into two steps:
  • 1: Voting
    • Each participant prepares to commit and votes of whether or not it can commit
2-Phase Commit

- Separate the commit into two steps:
  - 1: Voting
    - Each participant prepares to commit and votes of whether or not it can commit
  - 2: Committing
    - Once voting succeeds, every participant commits or aborts
2PC: Voting
2PC: Voting

• Coordinator asks each participant: can you commit for this transaction?
2PC: Voting

- Coordinator asks each participant: can you commit for this transaction?
- Each participant prepares to commit BEFORE answering yes
2PC: Voting

• Coordinator asks each participant: can you commit for this transaction?

• Each participant prepares to commit BEFORE answering yes
  • e.g. save transaction to disk for later recovery
2PC: Voting

• Coordinator asks each participant: can you commit for this transaction?

• Each participant prepares to commit BEFORE answering yes
  • e.g. save transaction to disk for later recovery
  • Can not abort after saying yes
2PC: Voting

- Coordinator asks each participant: can you commit for this transaction?
- Each participant prepares to commit BEFORE answering yes
  - e.g. save transaction to disk for later recovery
  - Can not abort after saying yes
- Outcome of transaction is unknown until the coordinator receives all votes and says “do abort” or “do commit”
2PC Event Sequence

Coordinator
Transaction state:
prepared

Participant
Local state:
prepared
2PC Event Sequence

Coordinator
Transaction state: prepared

Participant
Local state: prepared uncertain

Can you commit?
2PC Event Sequence

Coordinator

Transaction state: prepared

Participant

Local state: prepared

Can you commit?

Yes

uncertain
2PC Event Sequence

Coordinator
Transaction state:
- \textit{prepared}
- \textit{committed}

Can you commit?
- Yes
- OK, commit

Participant
Local state:
- \textit{prepared}
- \textit{uncertain}
- \textit{committed}
2PC Event Sequence

Coordinator

Transaction state:
- prepared
- committed
- done

Can you commit?
- Yes
- OK, commit
- OK I committed

Participant

Local state:
- prepared
- uncertain
- committed
2PC Example
2PC Example

transferMoney(“from”: Barney@Goliath National, “to”: Mortimer@ Duke&Duke, “amount”=$1)

Initially: Barney.balance= $10000, Mortimer.balance=$10000
2PC Example

```javascript
transferMoney("from": Barney@Goliath National, "to": Mortimer@ Duke&Duke, "amount"=$1)
```

Initially: Barney.balance= $10000, Mortimer.balance=$10000

Requirements:
1. Atomicity (transfer happens or doesn’t)
2. Concurrency control (serializability)
2PC Example

For simplicity, let’s assume transfer is:

```c
int transfer(src, dst, amt) {
    transaction = begin();
    src.bal -= amt;
    dst.bal += amt;
    return transaction.commit();
}
```
2PC Example

Coordinator (client or 3rd party)

transaction .commit()

Participant Goliath National

Participant Duke & Duke
2PC Example

Coordinator (client or 3rd party)  
transaction.commit()

Participant Goliath National

Participant Duke & Duke

prepare

prepare
2PC Example

Coordinator (client or 3rd party)

Participant Goliath National

Participant Duke & Duke

transaction.commit()

prepare

prepare

If we can commit, then lock our customer, vote "yes"
2PC Example

Coordinator (client or 3rd party)  
Transaction .commit()

Participant Goliath National

Participant Duke & Duke

If we can commit, then lock our customer, vote “yes”
2PC Example

Transaction

Coordinator (client or 3rd party)

Participant Goliath National

Participant Duke & Duke

If we can commit, then lock our customer, vote “yes”

If everyone can commit, then outcome == commit, else abort

transaction.commit()
2PC Example

Coordinator (client or 3rd party)  Participant Goliath National  Participant Duke & Duke

transaction.commit()

prepare

response_{GNB}

response_{D&D}

prepare

outcome

outcome

If we can commit, then lock our customer, vote “yes”

If everyone can commit, then outcome == commit, else abort
Fault Recovery

• How do we recover transaction state if we crash?

• Goal:
  • Committed transactions are not lost
  • Non-committed transactions either continue where they were or aborted

• Plan:
  • Consider local recovery
  • Then distributed issues
Write-ahead logging

• Maintain a complete log of all operations INDEPENDENT of the actual data they apply to
  • E.g. Transaction boundaries and updates
• Transaction operations considered provisional until commit is logged to disk
  • Log is authoritative
Write ahead logging: Begin/commit/abort

- Maintain this big log, with…
- Log Sequence Numbers (LSN) to track entries
- Each record contains an LSN, plus the LSN of the previous transaction
- Transaction ID
- Operation type
Write ahead logging: update records

• Track all information needed to reproduce transaction
  • prevLSN, transactionID, operationType (like begin/commit/abort)
  • Update itself:
    • Update location
    • Old value
    • New value
Recovering From Failure
Recovering From Failure

• Let’s assume we can always read the log
Recovering From Failure

• Let’s assume we can always read the log
• Analyze the log
Recovering From Failure

- Let’s assume we can always read the log
- Analyze the log
- Redo all transactions starting from beginning
Recovering From Failure

• Let’s assume we can always read the log
• Analyze the log
• Redo all transactions starting from beginning
• Undo uncommitted transactions
Recovering From Failure

- Let’s assume we can always read the log
- Analyze the log
- Redo all transactions starting from beginning
- Undo uncommitted transactions
  - We replay all of the transactions for consistency
Recovering From Failure

- Let’s assume we can always read the log
- Analyze the log
- Redo all transactions starting from beginning
- Undo uncommitted transactions
  - We replay all of the transactions for consistency
  - Generalize all operations - don’t need to store the results of operations, just the operations
Write Ahead Logging + Checkpoints
Write Ahead Logging + Checkpoints

• If you have a checkpoint, you can guarantee that all things before that checkpoint have been flushed to disk
Write Ahead Logging + Checkpoints

• If you have a checkpoint, you can guarantee that all things before that checkpoint have been flushed to disk

• Hence, no need to replay log after then
Write Ahead Logging + Checkpoints

- If you have a checkpoint, you can guarantee that all things before that checkpoint have been flushed to disk
- Hence, no need to replay log after then
- Speeds up recovery
Write Ahead Logging + Checkpoints

- If you have a checkpoint, you can guarantee that all things before that checkpoint have been flushed to disk
- Hence, no need to replay log after then
- Speeds up recovery
- Reduces log size
Write Ahead Logging + Checkpoints

• If you have a checkpoint, you can guarantee that all things before that checkpoint have been flushed to disk

• Hence, no need to replay log after then

• Speeds up recovery

• Reduces log size

• Can always build one checkpoint off an old one
Write Ahead Logging + Checkpoints

• If you have a checkpoint, you can guarantee that all things before that checkpoint have been flushed to disk
• Hence, no need to replay log after then
• Speeds up recovery
• Reduces log size
• Can always build one checkpoint off an old one
• Why not always checkpoint?
Recovery in 2PC
Recovery in 2PC

• What to log?
  • State changes in protocol
  • Participants: prepared; uncertain; committed/aborted
  • Coordinator: prepared; committed/aborted; done
  • These messages are idempotent - can be repeated
Recovery in 2PC

• What to log?
  • State changes in protocol
  • Participants: prepared; uncertain; committed/aborted
  • Coordinator: prepared; committed/aborted; done
  • These messages are idempotent - can be repeated

• Recovery depends on failure
  • Crash + reboot + recover
  • Timeout + recover
Crash + Reboot Recovery
Crash + Reboot Recovery

- Nodes can’t back out once commit is decided
Crash + Reboot Recovery

- Nodes can’t back out once commit is decided
- If coordinator crashes just AFTER deciding “commit”
  - Must remember this decision, replay
Crash + Reboot Recovery

- Nodes can’t back out once commit is decided
- If coordinator crashes just AFTER deciding “commit”
  - Must remember this decision, replay
- If participant crashes after saying “yes, commit”
  - Must remember this decision, replay
Crash + Reboot Recovery

• Nodes can’t back out once commit is decided

• If coordinator crashes just AFTER deciding “commit”
  • Must remember this decision, replay

• If participant crashes after saying “yes, commit”
  • Must remember this decision, replay

• Hence, all nodes need to log their progress in the protocol
2PC Example with logging

Coordinator (client or 3rd party)

Participant Goliath National

Participant Duke & Duke

transaction
.commit()
2PC Example with logging

Coordinator (client or 3rd party)

Participant Goliath National

Participant Duke & Duke

transaction.commit() Log!

Log!
2PC Example with logging

Coordinator (client or 3rd party)
Goliath National
Duke & Duke

transaction
.commit()

Prepare

Log!
2PC Example with logging

Coordinator (client or 3rd party)  
Participant Goliath National  
Participant Duke & Duke

transaction.commit()  
Log!  
prepare

Log!

Log!
2PC Example with logging

Coordinator (client or 3rd party) 

Participant 
Goliath National

Participant 
Duke & Duke

transaction .commit()

Log!

prepare

Log!

If we can commit, then lock our customer, vote “yes”
2PC Example with logging

Coordinator (client or 3rd party)

Participant Goliath National

Participant Duke & Duke

transaction .commit()

Log!

Log!

Log!

prepare

prepare

prepare

response_{GNB}

response_{D&D}

If we can commit, then lock our customer, vote “yes”
2PC Example with logging

Coordinator (client or 3rd party)  Participant Goliath National  Participant Duke & Duke

transaction.commit()  prepare  prepare

Log!  response_{GNB}  response_{D&D}

Log!  Log!

If we can commit, then lock our customer, vote “yes”

If everyone can commit, then outcome == commit, else abort
2PC Example with logging

Coordinator (client or 3rd party)
transaction .commit()

Participant Goliath National
prepare
response\textsubscript{GNB}

Participant Duke & Duke
prepare
response\textsubscript{D\&D}

Log!

Log!
Log!

If we can commit, then lock our customer, vote “yes”

If everyone can commit, then outcome == commit, else abort
2PC Example with logging

transaction.commit()  
Coordinator (client or 3rd party)

Log!

Participant Goliath National

Log!

Participant Duke & Duke

Log!

If we can commit, then lock our customer, vote “yes”

If everyone can commit, then outcome == commit, else abort
2PC Example with logging

Coordinator (client or 3rd party)  Participant Goliath National  Participant Duke & Duke

transaction.commit()  prepare  prepare

Log!  response_{GNB}  response_{D&D}  Log!

Log!  outcome  outcome

If we can commit, then lock our customer, vote “yes”

If everyone can commit, then outcome == commit, else abort

Log!
Recovery on Reboot
Recovery on Reboot

• If coordinator finds no “commit” message on disk, abort
Recovery on Reboot

• If coordinator finds no “commit” message on disk, abort
• If coordinator finds “commit” message, commit
Recovery on Reboot

• If coordinator finds no “commit” message on disk, abort
• If coordinator finds “commit” message, commit
• If participant finds no “yes, ok” message, abort
Recovery on Reboot

• If coordinator finds no “commit” message on disk, abort
• If coordinator finds “commit” message, commit
• If participant finds no “yes, ok” message, abort
• If participant finds “yes, ok” message, then replay that message and continue protocol
Timeouts in 2PC
Timeouts in 2PC

• Example:
  • Coordinator times out waiting for Goliath National Bank’s response
  • Bank times out waiting for coordinator’s outcome message
Timeouts in 2PC

• Example:
  • Coordinator times out waiting for Goliath National Bank’s response
  • Bank times out waiting for coordinator’s outcome message

• Causes?
  • Network
  • Overloaded hosts
  • Both are very realistic…
Coordinator Timeouts
Coordinator Timeouts

• If coordinator times out waiting to hear from a bank
Coordinator Timeouts

• If coordinator times out waiting to hear from a bank
  • Coordinator hasn’t sent any commit messages yet
Coordinator Timeouts

• If coordinator times out waiting to hear from a bank
  • Coordinator hasn’t sent any commit messages yet
  • Can safely abort - send abort message
Coordinator Timeouts

• If coordinator times out waiting to hear from a bank
  • Coordinator hasn’t sent any commit messages yet
  • Can safely abort - send abort message
  • Preserves correctness, sacrifices performance (maybe didn’t need to abort!)
Coordinator Timeouts

- If coordinator times out waiting to hear from a bank
  - Coordinator hasn’t sent any commit messages yet
  - Can safely abort - send abort message
  - Preserves correctness, sacrifices performance (maybe didn’t need to abort!)
- If either bank decided to commit, it’s fine - they will eventually abort
Handling Bank Timeouts
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?

• If bank voted “no”, it’s OK to abort
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?
• If bank voted “no”, it’s OK to abort
• If bank voted “yes”
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?

• If bank voted “no”, it’s OK to abort

• If bank voted “yes”
  • It can’t decide to abort (maybe both banks voted “yes” and coordinator heard this)
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?

• If bank voted “no”, it’s OK to abort

• If bank voted “yes”
  • It can’t decide to abort (maybe both banks voted “yes” and coordinator heard this)
  • It can’t decide to commit (maybe other bank voted yes)
Handling Bank Timeouts

• What if the bank doesn’t hear back from coordinator?
• If bank voted “no”, it’s OK to abort
• If bank voted “yes”
  • It can’t decide to abort (maybe both banks voted “yes” and coordinator heard this)
  • It can’t decide to commit (maybe other bank voted yes)
• Does bank just wait for ever?
Handling Bank Timeouts
Handling Bank Timeouts

• Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit
Handling Bank Timeouts

• Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit

• Bank asks other bank for status (if it heard from coordinator)
Handling Bank Timeouts

• Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit

• Bank asks other bank for status (if it heard from coordinator)

• If other bank heard “commit” or “abort” then do that
Handling Bank Timeouts

- Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit
- Bank asks other bank for status (if it heard from coordinator)
- If other bank heard “commit” or “abort” then do that
- If other bank didn’t hear
Handling Bank Timeouts

• Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit

• Bank asks other bank for status (if it heard from coordinator)

• If other bank heard “commit” or “abort” then do that

• If other bank didn’t hear
  • but other voted “no”: both banks abort
Handling Bank Timeouts

• Can resolve SOME timeout problems with guaranteed correctness in event bank voted “yes” to commit

• Bank asks other bank for status (if it heard from coordinator)

• If other bank heard “commit” or “abort” then do that

• If other bank didn’t hear
  • but other voted “no”: both banks abort
  • but other voted “yes”: no decision possible!
2PC Timeouts
2PC Timeouts

• We can solve a lot (but not all of the cases) by having the participants talk to each other
2PC Timeouts

• We can solve a lot (but not all of the cases) by having the participants talk to each other
• But, if coordinator fails, there are cases where everyone stalls until it recovers
2PC Timeouts

• We can solve a lot (but not all of the cases) by having the participants talk to each other

• But, if coordinator fails, there are cases where everyone stalls until it recovers

• Can the coordinator fail? … yes
2PC Timeouts

- We can solve a lot (but not all of the cases) by having the participants talk to each other.
- But, if coordinator fails, there are cases where everyone stalls until it recovers.
- Can the coordinator fail? … yes.
- We’ll come back to this “discuss amongst yourselves” kind of transactions next week.
Lab 5: Lock Server

- We’re going to build, together, a lock server like from HW2
- Address book example
  - Lock server allows read/write locks on entire list
  - Lock server will allow an entry to be locked by exactly one client at a time
  - Clients will track which locks they have
Lab 5: Lock Server
Semantics

- List contents of address book: read lock on list
- Add new entry: write lock on list
- Update entry: write lock on the entry