Transactions & Concurrency

Yao Liu
Transactions

• Motivation
  - Provide atomic operations at servers that maintain shared data for clients
  - Provide recoverability from server crashes

• Properties
  - Atomicity, Consistency, Isolation, Durability (ACID)

• Concepts: commit, abort
Operations of the **Account** interface

- **deposit(amount)**
  - deposit amount in the account
- **withdraw(amount)**
  - withdraw amount from the account
- **getBalance() -> amount**
  - return the balance of the account
- **setBalance(amount)**
  - set the balance of the account to amount

---

Operations of the **Branch** interface

- **create(name) -> account**
  - create a new account with a given name
- **lookUp(name) -> account**
  - return a reference to the account with the given name
- **branchTotal() -> amount**
  - return the total of all the balances at the branch
A client’s banking transaction

Transaction $T$:
\begin{align*}
a &. \text{withdraw}(100); \\
b &. \text{deposit}(100); \\
c &. \text{withdraw}(200); \\
b &. \text{deposit}(200);
\end{align*}
Operations in Coordinator interface

\textit{openTransaction() -> trans;}
starts a new transaction and delivers a unique TID \textit{trans}. This identifier will be used in the other operations in the transaction.

\textit{closeTransaction(trans) -> (commit, abort);} 
ends a transaction: a \textit{commit} return value indicates that the transaction has committed; an \textit{abort} return value indicates that it has aborted.

\textit{abortTransaction(trans);}  
aborts the transaction.
## Transaction life histories

<table>
<thead>
<tr>
<th>Successful</th>
<th>Aborted by client</th>
<th>Aborted by server</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>openTransaction</code></td>
<td><code>openTransaction</code></td>
<td><code>openTransaction</code></td>
</tr>
<tr>
<td><code>operation</code></td>
<td><code>operation</code></td>
<td><code>operation</code></td>
</tr>
<tr>
<td><code>operation</code></td>
<td><code>operation</code></td>
<td><code>operation ERROR</code></td>
</tr>
<tr>
<td><code>operation</code></td>
<td><code>operation</code></td>
<td><code>operation ERROR</code></td>
</tr>
<tr>
<td><code>operation</code></td>
<td><code>operation</code></td>
<td><code>operation ERROR</code></td>
</tr>
<tr>
<td><code>closeTransaction</code></td>
<td><code>abortTransaction</code></td>
<td><code>server aborts transaction</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>server aborts transaction</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>server aborts transaction</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>server aborts transaction</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>server aborts transaction</code></td>
</tr>
</tbody>
</table>

The table above illustrates the possible life histories of a transaction. The first column shows the successful operations, the middle column shows the operations that are aborted by the client, and the last column shows the operations that are aborted by the server. The server aborts the transaction if an error occurs, and the error is reported to the client.
Concurrency control

• Motivation: without concurrency control, we have lost updates, inconsistent retrievals, dirty reads, etc. (see following slides)
• Concurrency control schemes are designed to allow two or more transactions to be executed correctly while maintaining serial equivalence
  • Serial Equivalence is correctness criterion
    • Schedule produced by concurrency control scheme should be equivalent to a serial schedule in which transactions are executed one after the other
• Schemes: locking, optimistic concurrency control, timestamp based concurrency control
The lost update problem

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b.getBalance();$</td>
<td>$balance = b.getBalance();$</td>
</tr>
<tr>
<td>$b.setBalance(balance*1.1);$</td>
<td>$b.setBalance(balance*1.1);$</td>
</tr>
<tr>
<td>$a.withdraw(balance/10)$</td>
<td>$c.withdraw(balance/10)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$balance = b.getBalance();$</th>
<th>$200$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b.setBalance(balance*1.1);$</td>
<td>$220$</td>
</tr>
<tr>
<td>$a.withdraw(balance/10)$</td>
<td>$80$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$balance = b.getBalance();$</th>
<th>$200$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b.setBalance(balance*1.1);$</td>
<td>$220$</td>
</tr>
<tr>
<td>$c.withdraw(balance/10)$</td>
<td>$280$</td>
</tr>
</tbody>
</table>
The inconsistent retrievals problem

<table>
<thead>
<tr>
<th>Transaction V</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100)</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(100);</td>
<td>total = a.getBalance() $100</td>
</tr>
<tr>
<td>a.withdraw(100);</td>
<td>total = total+b.getBalance() $300</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>total = total+c.getBalance()</td>
</tr>
<tr>
<td></td>
<td>$300</td>
</tr>
</tbody>
</table>
A serially equivalent interleaving of $T$ and $U$

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Transaction $U$</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance()</td>
<td>balance = b.getBalance()</td>
</tr>
<tr>
<td>b.setBalance(balance*1.1)</td>
<td>b.setBalance(balance*1.1)</td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>c.withdraw(balance/10)</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>balance = b.getBalance()</td>
<td>balance = b.getBalance()</td>
</tr>
<tr>
<td>$200$</td>
<td>$220$</td>
</tr>
<tr>
<td>b.setBalance(balance*1.1)</td>
<td>b.setBalance(balance*1.1)</td>
</tr>
<tr>
<td>$220$</td>
<td>$242$</td>
</tr>
<tr>
<td>a.withdraw(balance/10)</td>
<td>c.withdraw(balance/10)</td>
</tr>
<tr>
<td>$80$</td>
<td>$278$</td>
</tr>
</tbody>
</table>
A serially equivalent interleaving of V and W

<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100);</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(100);</td>
<td>total = a.getBalance()</td>
</tr>
<tr>
<td>b.deposit(100)</td>
<td>total = total + b.getBalance()</td>
</tr>
<tr>
<td></td>
<td>total = total + c.getBalance()</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

$100 \quad $300

$100 \quad $400
Serializability

BEGIN TRANSACTION
x = 0;
x = x + 1;
END_TRANSACTION

(a)

BEGIN TRANSACTION
x = 0;
x = x + 2;
END_TRANSACTION

(b)

BEGIN TRANSACTION
x = 0;
x = x + 3;
END_TRANSACTION

(c)

Schedule 1
x = 0;  x = x + 1;  x = 0;  x = x + 2;  x = 0;  x = x + 3
Legal

Schedule 2
x = 0;  x = 0;  x = x + 1;  x = x + 2;  x = 0;  x = x + 3;
Legal

Schedule 3
x = 0;  x = 0;  x = x + 1;  x = 0;  x = x + 2;  x = x + 3;
Illegal

(d)

a) – c) Three transactions $T_1$, $T_2$, and $T_3$
d) Possible schedules
# Read and write operation conflict rules

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read read</td>
<td>No</td>
<td>Because the effect of a pair of <em>read</em> operations does not depend on the order in which they are executed</td>
</tr>
<tr>
<td>read write</td>
<td>Yes</td>
<td>Because the effect of a <em>read</em> and a <em>write</em> operation depends on the order of their execution</td>
</tr>
<tr>
<td>write write</td>
<td>Yes</td>
<td>Because the effect of a pair of <em>write</em> operations depends on the order of their execution</td>
</tr>
</tbody>
</table>
A non-serially equivalent interleaving of operations of transactions T and U

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \text{read}(i)$</td>
<td>$y = \text{read}(j)$</td>
</tr>
<tr>
<td>$\text{write}(i, 10)$</td>
<td>$\text{write}(j, 30)$</td>
</tr>
<tr>
<td>$\text{write}(j, 20)$</td>
<td>$z = \text{read}(i)$</td>
</tr>
</tbody>
</table>
A dirty read when transaction $T$ aborts

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$.getBalance()</td>
<td>$a$.getBalance()</td>
</tr>
<tr>
<td>$a$.setBalance(balance + 10)</td>
<td>$a$.setBalance(balance + 20)</td>
</tr>
<tr>
<td>$balance = a$.getBalance() $100$</td>
<td>$balance = a$.getBalance() $110$</td>
</tr>
<tr>
<td>$a$.setBalance(balance + 10) $110$</td>
<td>$a$.setBalance(balance + 20) $130$</td>
</tr>
<tr>
<td>abort transaction</td>
<td>commit transaction</td>
</tr>
</tbody>
</table>
Implementing transactions: private workspace

a) The file index and disk blocks for a three-block file
b) The situation after a transaction has modified block 0 and appended block 3
c) After committing
Implementing transactions: write-ahead log

a) A transaction

b) – d) The log before each statement is executed

```plaintext
x = 0;
y = 0;
BEGIN_TRANSACTION;
x = x + 1;
y = y + 2
x = y * y;
END_TRANSACTION;
```

Log

Log

Log

\[x = 0 / 1\]

\[x = 0 / 1\]

\[x = 0 / 1\]

\[y = 0 / 2\]

\[y = 0 / 2\]

\[x = 1 / 4\]
Concurrency control

General organization of managers for handling transactions.
## Transactions $T$ and $U$ with exclusive locks

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th></th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$balance = b \cdot getBalance()$</td>
<td>$balance = b \cdot getBalance()$</td>
<td></td>
</tr>
<tr>
<td>$b \cdot setBalance(bal*1.1)$</td>
<td>$b \cdot setBalance(bal*1.1)$</td>
<td></td>
</tr>
<tr>
<td>$a \cdot withdraw(bal/10)$</td>
<td>$c \cdot withdraw(bal/10)$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operations</th>
<th>Locks</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td></td>
<td>openTransaction</td>
<td>waits for $T$’s lock on $B$</td>
</tr>
<tr>
<td>$bal = b \cdot getBalance()$</td>
<td>lock $B$</td>
<td>$bal = b \cdot getBalance()$</td>
<td></td>
</tr>
<tr>
<td>$b \cdot setBalance(bal*1.1)$</td>
<td></td>
<td></td>
<td>lock $B$</td>
</tr>
<tr>
<td>$a \cdot withdraw(bal/10)$</td>
<td>lock $A$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>closeTransaction</td>
<td>unlock $A$, $B$</td>
<td>$b \cdot setBalance(bal*1.1)$</td>
<td>lock $C$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c \cdot withdraw(bal/10)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>closeTransaction</td>
<td>unlock $B$, $C$</td>
</tr>
</tbody>
</table>
## Lock compatibility

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock already set</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>none</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>read</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait</td>
</tr>
</tbody>
</table>

|                | write            | OK             |
|                | read             | wait           |
|                | write            | wait           |
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.
Two-Phase Locking (1)

- Two-phase locking

![Diagram showing Locking phases](Image)
Strict Two-Phase Locking (2)

- Strict two-phase locking.

![Diagram showing Strict Two-Phase Locking](image)

- Growing phase
- Shrinking phase
- Lock point
- All locks are released at the same time
# Deadlock with write locks

<table>
<thead>
<tr>
<th>Transaction $T$</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction $U$</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>$a.deposit(100)$</td>
<td>write lock $A$</td>
<td>Operations</td>
<td>$b.deposit(200)$</td>
<td>write lock $B$</td>
</tr>
<tr>
<td>$b.withdraw(100)$</td>
<td>waits for $U$'s lock on $B$</td>
<td></td>
<td>$a.withdraw(200)$;</td>
<td>waits for $T$'s lock on $A$</td>
<td></td>
</tr>
</tbody>
</table>
The wait-for graph
A cycle in a wait-for graph
Another wait-for graph

T/U/V share a read lock on object C.
W holds a write lock on B, on which V is waiting to obtain a lock.
T/W then request write locks on object C.
Deadlock: T waits for U/V, V waits for W, W waits for T/U/V.
Resolution of deadlock

<table>
<thead>
<tr>
<th>Transaction T</th>
<th>Operations</th>
<th>Locks</th>
<th>Transaction U</th>
<th>Operations</th>
<th>Locks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(a.\text{deposit}(100));</td>
<td>write lock (A)</td>
<td>(b.\text{deposit}(200))</td>
<td>write lock (B)</td>
<td></td>
</tr>
<tr>
<td>(b.\text{withdraw}(100))</td>
<td>waits for (U)'s</td>
<td></td>
<td>(a.\text{withdraw}(200);)</td>
<td>waits for (T)'s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lock on (B)</td>
<td></td>
<td></td>
<td>lock on (A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(timeout elapses)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(T)'s lock on (A) becomes vulnerable,</td>
<td>unlock (A), abort (T)</td>
<td>(a.\text{withdraw}(200);)</td>
<td>write locks (A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unlock (A), abort (T)</td>
<td></td>
<td></td>
<td>unlock (A, B)</td>
<td></td>
</tr>
</tbody>
</table>
Optimistic concurrency control

- **Drawbacks of locking**
  - Overhead of lock maintenance
  - Deadlocks
  - Reduced concurrency

- **Optimistic Concurrency Control**
  - In most applications, likelihood of conflicting accesses by concurrent transactions is low
  - Transactions proceed as though there are no conflicts
  - Three phases
    - Working Phase – transactions read and write private copies of objects
    - Validation Phase – each transaction is assigned a transaction number when it enters this phase
    - Update Phase
Optimistic concurrency control: serializability of transaction $T_v$ w.r.t. $T_i$

$T_v$ and $T_i$ are overlapping transactions.

For $T_v$ to be serializable w.r.t $T_i$ the following rules must hold:

<table>
<thead>
<tr>
<th>$T_v$</th>
<th>$T_i$</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>write</td>
<td>read</td>
<td>1. $T_i$ must not read objects written by $T_v$</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>2. $T_v$ must not read objects written by $T_i$</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>3. $T_i$ must not write objects written by $T_v$ and $T_v$ must not write objects written by $T_i$</td>
</tr>
</tbody>
</table>

If simplification is made that only one transaction may be in its validation or write phases at one time, then third rule is always satisfied.
Validation of transactions

T₁ Working Validation Update

T₂

T₃

Transaction being validated

Tᵥ

Earlier committed transactions

Later active transactions

active₁

active₂
Validation of transactions

Backward validation of transaction $T_v$

boolean valid = true;
for (int $T_i = startTn+1; T_i <= finishTn; T_i++){
    if (read set of $T_v$ intersects write set of $T_i$) valid = false;
}

Forward validation of transaction $T_v$

boolean valid = true;
for (int $T_{id} = active1; T_{id} <= activeN; T_{id}++){
    if (write set of $T_v$ intersects read set of $T_{id}$) valid = false;
}
Timestamp based concurrency control

• Each transaction is assigned a unique timestamp at the moment it starts
  • In distributed transactions, Lamport timestamps can be used

• Every data item has a timestamp
  • Read timestamp = timestamp of transaction that last read the item
  • Write timestamp = timestamp of transaction that most recently changed an item
Operation conflicts for timestamp ordering

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>write</td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>$T_c$ must not write an object that has been read by any $T_i$ where $T_i &gt; T_c$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>this requires that $T_c \geq$ the maximum read timestamp of the object.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>write</td>
<td>write</td>
</tr>
<tr>
<td></td>
<td>$T_c$ must not write an object that has been written by any $T_i$ where $T_i &gt; T_c$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>this requires that $T_c &gt;$ write timestamp of the committed object.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td></td>
<td>$T_c$ must not read an object that has been written by any $T_i$ where $T_i &gt; T_c$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>this requires that $T_c &gt;$ write timestamp of the committed object.</td>
<td></td>
</tr>
</tbody>
</table>
Timestamp ordering write rule

if ($T_c \geq$ maximum read timestamp on $D$ && $T_c >$ write timestamp on committed version of $D$)
    perform write operation on tentative version of $D$ with write timestamp $T_c$
else /* write is too late */
    Abort transaction $T_c$
Write operations and timestamps

(a) $T_3$ write

Before

$T_2$

After

$T_2$ $T_3$

(b) $T_3$ write

Before

$T_1$ $T_2$

After

$T_1$ $T_2$ $T_3$

(c) $T_3$ write

Before

$T_1$ $T_4$

After

$T_1$ $T_3$ $T_4$

(d) $T_3$ write

Before

$T_4$

After

$T_4$

Key:

- Tentative
- Committed

Object produced by transaction $T_i$ (with write timestamp $T_i$)

$T_1 < T_2 < T_3 < T_4$
Timestamp ordering read rule

if ( $T_c >$ write timestamp on committed version of $D$) {
    let $D_{selected}$ be the version of $D$ with the maximum write timestamp $\leq T_c$
    if ($D_{selected}$ is committed)
        perform read operation on the version $D_{selected}$
    else
        Wait until the transaction that made version $D_{selected}$ commits or aborts
        then reapply the read rule
} else
    Abort transaction $T_c$
Read operations and timestamps

(a) $T_3$ read

(b) $T_3$ read

(c) $T_3$ read

(d) $T_3$ read

Key:
- Committed
- Tentative

Object produced by transaction $T_i$ (with write timestamp $T_i$)
$T_1 < T_2 < T_3 < T_4$
### Timestamps in transactions \( T \) and \( U \)

<table>
<thead>
<tr>
<th></th>
<th>( T )</th>
<th>( U )</th>
<th>Timestamps and versions of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>( openTransaction )</td>
<td>bal = b.getBalance()</td>
<td>bal = b.getBalance()</td>
<td>B</td>
</tr>
</tbody>
</table>
| \( b.setBalance(bal*1.1) \) | | | A  
| \( a.withdraw(bal/10) \) | | | C  
| \( commit \) | | | S, T  
| \( b.setBalance(bal*1.1) \) | | | \{T\}  
| \( c.withdraw(bal/10) \) | | | S  

Timestamps of committed transactions are in **BOLD**
Distributed Transactions
Distributed transactions

(a) Flat transaction

(b) Nested transactions
Nested banking transaction

\[ T = \text{openTransaction} \]

\[
\begin{align*}
\text{openSubTransaction} & \quad a.\text{withdraw}(10) \\
\text{openSubTransaction} & \quad b.\text{withdraw}(20) \\
\text{openSubTransaction} & \quad c.\text{deposit}(10) \\
\text{openSubTransaction} & \quad d.\text{deposit}(20) \\
\text{closeTransaction} & 
\end{align*}
\]
A distributed banking transaction

$T = \text{openTransaction}$

- \text{a.withdraw}(4);
- \text{c.deposit}(4);
- \text{b.withdraw}(3);
- \text{d.deposit}(3);

\text{closeTransaction}

Note: the coordinator is in one of the servers, e.g., BranchX
Concurrence control for distributed transactions

General organization of managers for handling distributed transactions.
Concurrency control for distributed transactions

- **Locking**
  - Distributed deadlocks possible

- **Timestamp ordering**
  - Lamport timestamps
    - for efficiency it is required that timestamps issued by coordinators be roughly synchronized
### Interleavings of transactions U, V and W

<table>
<thead>
<tr>
<th>U</th>
<th>V</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>d.deposit(10) lock D</td>
<td>b.deposit(10) lock B at Y</td>
<td>c.deposit(30) lock C at Z</td>
</tr>
<tr>
<td>a.deposit(20) lock A at X</td>
<td>c.withdraw(20) wait at Z</td>
<td>a.withdraw(20) wait at X</td>
</tr>
<tr>
<td>b.withdraw(30) wait at Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed deadlock

(a)

(b)
Local and global wait-for graphs

**Phantom deadlock:** when the deadlock detector collects wait-for graphs from each participant, some locks may be released during the course. Suppose U releases an object at server X and requests the one held by V at Y. If global detector receives Y’s local graph before server X’s, it would detect a cycle T->U->V->T, although T->U no longer exists.
Atomic commit protocols

• The atomicity of a transaction requires that when a distributed transaction comes to an end, either all of its operations are carried out or none of them

• One phase commit
  • Coordinator tells all participants to commit
    • If a participant cannot commit (say because of concurrency control), no way to inform coordinator

• Two phase commit (2PC)
The two-phase commit protocol

Phase 1 (voting phase):
1. The coordinator sends a `canCommit?` request to each of the participants in the transaction.
2. When a participant receives a `canCommit?` request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

Phase 2 (completion according to outcome of vote):
3. The coordinator collects the votes (including its own).
   (a) If there are no failures and all the votes are Yes the coordinator decides to commit the transaction and sends a `doCommit` request to each of the participants.
   (b) Otherwise the coordinator decides to abort the transaction and sends `doAbort` requests to all participants that voted Yes.
4. Participants that voted Yes are waiting for a `doCommit` or `doAbort` request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a `haveCommitted` call as confirmation to the coordinator.
Operations for two-phase commit protocol

```
canCommit?(trans) -> Yes / No
    Call from coordinator to participant to ask whether it can commit a
    transaction. Participant replies with its vote.

doCommit(trans)
    Call from coordinator to participant to tell participant to commit its part of a
    transaction.

doAbort(trans)
    Call from coordinator to participant to tell participant to abort its part of a
    transaction.

haveCommitted(trans, participant)
    Call from participant to coordinator to confirm that it has committed the
    transaction.

getDecision(trans) -> Yes / No
    Call from participant to coordinator to ask for the decision on a transaction
    after it has voted Yes but has still had no reply after some delay. Used to
    recover from server crash or delayed messages.
```
Communication in two-phase commit protocol

**Coordinator**

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>prepared to commit</td>
</tr>
<tr>
<td></td>
<td>(waiting for votes)</td>
</tr>
<tr>
<td>3</td>
<td>committed</td>
</tr>
<tr>
<td></td>
<td>done</td>
</tr>
</tbody>
</table>

**Participant**

<table>
<thead>
<tr>
<th>step</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>prepared to commit</td>
</tr>
<tr>
<td></td>
<td>(uncertain)</td>
</tr>
<tr>
<td>4</td>
<td>committed</td>
</tr>
</tbody>
</table>

canCommit? Yes

doCommit

haveCommitted
Two-Phase Commit (1)

a) The finite state machine for the coordinator in 2PC.
b) The finite state machine for a participant.
## Two-Phase Commit (2)

<table>
<thead>
<tr>
<th>State of Q</th>
<th>Action by P</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMIT</td>
<td>Make transition to COMMIT</td>
</tr>
<tr>
<td>ABORT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>INIT</td>
<td>Make transition to ABORT</td>
</tr>
<tr>
<td>READY</td>
<td>Contact another participant</td>
</tr>
</tbody>
</table>

Actions taken by a participant $P$ when residing in state \textit{READY} and having contacted another participant $Q$. 
Two-Phase Commit (3)

actions by coordinator:

write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected {
    wait for any incoming vote;
    if timeout {
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
        exit;
    }
    record vote;
}
if all participants sent VOTE_COMMIT and coordinator votes COMMIT{
    write GLOBAL_COMMIT to local log;
multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT to local log;
multicast GLOBAL_ABORT to all participants;
}

Outline of the steps taken by the coordinator in a two phase commit protocol
Two-Phase Commit (4)

**actions by participant:**

write INIT to local log;
wait for VOTE_REQUEST from coordinator;
if timeout {
    write VOTE_ABORT to local log;
    exit;
}
if participant votes COMMIT {
    write VOTE_COMMIT to local log;
    send VOTE_COMMIT to coordinator;
    wait for DECISION from coordinator;
    if timeout {
        multicast DECISION_REQUEST to other participants;
        wait until DECISION is received; /* remain blocked */
        write DECISION to local log;
    }
    if DECISION == GLOBAL_COMMIT
        write GLOBAL_COMMIT to local log;
    else if DECISION == GLOBAL_ABORT
        write GLOBAL_ABORT to local log;
} else {
    write VOTE_ABORT to local log;
    send VOTE_ABORT to coordinator;
}
Two-Phase Commit (5)

**actions for handling decision requests:** /* executed by separate thread */

```c
while true {
    wait until any incoming DECISION_REQUEST is received; /* remain blocked */
    read most recently recorded STATE from the local log;
    if STATE == GLOBAL_COMMIT
        send GLOBAL_COMMIT to requesting participant;
    else if STATE == INIT or STATE == GLOBAL_ABORT
        send GLOBAL_ABORT to requesting participant;
    else
        skip; /* participant remains blocked */
```

Steps taken for handling incoming decision requests.
Three-Phase Commit

- **Problem with 2PC**
  - If coordinator crashes, participants cannot reach a decision, stay blocked until coordinator recovers

- **3PC**
  - There is no single state from which it is possible to make a transition directly to either COMMIT or ABORT states
  - There is no state in which it is not possible to make a final decision, and from which a transition to COMMIT can be made
Three-Phase Commit

(a) Finite state machine for the coordinator in 3PC

(b) Finite state machine for a participant
Reading

• Section 8.5 of Tbook
• Chapters 16 and 17 of Cbook