CIS 630 - Fall 2005
Distributed Systems

Lecture 7
Transactions and Concurrency Control

University of Oregon
Department of Computer and Information Science
Term project proposals due yesterday ✓
  ☑ Feedback give to everyone
  ☑ Let me know if you want more

Programming assignment due end of week ✓
  ☑ Demo time this afternoon, 4-5pm

Paper 2 summary
  ☑ Write for one of the two papers
  ☑ Read both papers
  ☑ Due on Friday, Oct. 28
Acknowledgements

- Some material taken from author’s teaching slides based on Distributed Systems: Concepts and Design book
- Some figures taken from Distributed Systems: Concepts and Design book
Lecture Objectives

☐ To study the design of services whose long-lived objects are intended to be shared by multiple clients

☐ To establish a model of a single-process server of long-lived objects

☐ To set up the all-orNothing and isolation properties of transactions within this model, so as to be able to study how they are maintained in the presence of concurrent clients and server failures

☐ To study the three main approaches to concurrency control, all of which maintain the isolation property of transactions in the presence of their concurrent execution at a single server
Introduction to Transactions

- Goal of transactions
  - Objects must remain in a consistent state
    - when they are accessed by multiple transactions
    - in the presence of server crashes

- Recoverable objects
  - Can be recovered after their server crashes
  - Objects are stored in permanent storage

- Failure model
  - Transactions deal with crash failures of processes and omission failures of communication

- Designed for an asynchronous system
  - It is assumed that messages may be delayed
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
- `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
Atomic Operations at Server

- Synchronization of client operations without transactions
- Server with multiple threads
  - Can perform several client operations concurrently
- If we allowed deposit and withdraw to run concurrently we could get inconsistent results
- Objects should be designed for safe concurrent access
  - In Java use synchronized methods
    
    ```java
    public synchronized void deposit(int amount) throws RemoteException
    ```
  - Effectively locks object until methods completes
- Atomic operations are free from interference from concurrent operations in other threads
- Use any available mutual exclusion mechanism on server
Client Cooperation by Synchronizing Servers

- Now consider clients share resources via a server
  - Some clients update server objects and others access them
- Servers with multiple threads require atomic objects
- In some applications, clients depend on one another to progress
  - Ex: one is a *producer* and another a *consumer*
  - Ex: one sets a lock and the other waits for it to be released
- Waiting client polls server to see if a resource is yet available
  - Potentially poor performance (why?) and unfair (why?)
- Java *wait* and *notify* methods allow threads to communicate with one another and to solve these problems
  - Ex: when a client requests a resource, the server thread waits until it is notified that the resource is available
  - Used within synchronized methods
Failure Model for Distributed Transactions

- Lampson [1981] proposed failure model
  - Deals with failures of disks, servers, communication
  - Algorithms work correctly when predictable faults

- Predictable faults
  - Writes to permanent storage may fail
    - reads can detect bad blocks by checksum
  - Servers may crash occasionally
    - recovery procedure to get its objects’ state
    - faulty servers are made to crash
  - A message delayed, lost, duplicated, or corrupted
    - detect corrupt messages (by checksum)
Transactions

- Some applications require a sequence of client requests to a server to be atomic in the sense that:
  - Free from interference other concurrent clients
  - Either all operations completed successfully or no effect at all in the presence of server crashes

- Consider transactions originating from database management systems

- Transactional file servers were built in the 1980s

- Transactions on distributed objects late 80s and 90s
  - Use middleware components

- Transactions apply to recoverable objects, are atomic

- Aim is maximize concurrency
A Client’s Banking Transaction

Transaction $T$:
\begin{itemize}
  \item $a$. withdraw(100);
  \item $b$. deposit(100);
  \item $c$. withdraw(200);
  \item $b$. deposit(200);
\end{itemize}

- Transaction specifies a sequence of related operations involving bank accounts named $A$, $B$ and $C$
  - Referred to as $a$, $b$ and $c$ in the program
- First two operations transfer $\$100$ from $A$ to $B$
- Second two operations transfer $\$200$ from $C$ to $B$
- $B$’s balance should be $\$300$ more
Atomicity of Transactions - Two Aspects

☐ All or nothing
  ● Either completes successfully and the effects of all of its operations are recorded in the objects
  ● Or (if it fails or is aborted) it has no effect at all
  ● Failure atomicity: effects are atomic even with crashes
  ● Durability: effects are saved in permanent storage
  ● Objects must be recoverable

☐ Isolation
  ● Each transaction performed without interference
    - no observation of a transaction's intermediate effects
  ● Concurrency control ensures isolation
  ● Must synchronize the operations sufficiently
ACID (Harder and Reuter [1983])

- Atomicity
- Consistency
  - Transaction take system from one consistent state to another
  - Generally programmer’s responsibility
- Isolation
- Durability
Operations in the *Coordinator* Interface

- Transaction capabilities may be added to a server of recoverable objects (assuming single server)
  - Each transaction is created and managed by a *Coordinator* object whose interface follows:

  \[
  \text{openTransaction}() -> \text{trans};
  \]
  
  starts a new transaction and delivers a unique TID \(\text{trans}\) to be used in the other operations in the transaction

  \[
  \text{closeTransaction}('\text{trans}') -> ('\text{commit}', '\text{abort}');
  \]
  
  ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted

  \[
  \text{abortTransaction}('\text{trans}');
  \]
  
  aborts the transaction
Transaction Operation

- A transaction is achieved by cooperation
  - Between client program
  - Recoverable objects
  - Coordinator

- Client specifies sequence of invocations on recoverable objects to comprise a transaction
  - Each transaction has an identifier (use in operations)

- Commits constitute an undertaking to the client that all of the changes requested are permanently recorded

- When a transaction aborts, the parties involved must ensure that none of its effects propagate to the future
## Transaction Life Histories

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

- A transaction is either successful (it commits)
  - Coordinator objects saved in permanent storage

- Or it is aborted by the client or the server
  - All temporary effects invisible to other transactions
  - How will client know when its transaction aborted?
    - next time it tries to access an object at the server
Concurrent Control

- Problems without appropriate concurrency control
  - *Lost update*
    - occurs when two transactions both read the old value of a variable and use it to calculate a new value
  - *Inconsistent retrievals*
    - occur when a retrieval transaction observes values that are involved in an ongoing updating transaction

- Serial equivalent executions of transactions can avoid

- Assume that the operations are synchronized
  - *deposit, withdraw, getBalance, setBalance*
  - Synchronized: effect on the account balance is atomic
The Lost Update Problem

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

- $balance = b\text{.getBalance}()$; $200$
- $b\text{.setBalance}(balance \times 1.1)$; $220$
- $a\text{.withdraw}(balance/10)$ $80$
- $c\text{.withdraw}(balance/10)$ $280$

- Problems?

- Initial balances:
  - $A - $100, $B - $200, $C - $300
- Both transactions increase $B$'s balance by 10%
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>$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>$300</td>
<td></td>
</tr>
</tbody>
</table>

- V transfers $100 from A to B while W calculates branch total (which should be $600)
Serial Equivalence

- Assume each one of a set of transactions has the correct effect when done on its own.
- Then if they are done one at a time in some order the effect will be correct.
- A *serially equivalent interleaving* is one in which the combined effect is the same as if the transactions had been done one at a time in some order.
- The same effect means (with respect to some order):
  - Read operations return the same values.
  - Instance variables of objects have same values at end.
- Transactions are scheduled to avoid access overlap.
### 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>

<table>
<thead>
<tr>
<th>$balance = b.getBalance()$</th>
<th>$balance = b.getBalance()$</th>
</tr>
</thead>
<tbody>
<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>

- If $T$ or $U$ runs before the other, can’t get lost update
- Also true if they are run in serially equivalent ordering
<table>
<thead>
<tr>
<th>Transaction V:</th>
<th>Transaction W:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.withdraw(100); b.deposit(100)</td>
<td>aBranch.branchTotal()</td>
</tr>
<tr>
<td>a.withdraw(100); b.deposit(100)</td>
<td>$100</td>
</tr>
<tr>
<td>$100</td>
<td>$300</td>
</tr>
<tr>
<td>$300</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>total = a.getBalance()</td>
<td>total = total+b.getBalance()</td>
</tr>
<tr>
<td>total = total+c.getBalance()</td>
<td>$400</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

- If $W$ run before or after $V$, the problem will not occur
- Will not occur in a serially equivalent ordering
- Illustration is serial, but it need not be (How?)
Serial Equivalence Criterion

- Adhering to serial equivalence prevents the occurrence of lost updates and inconsistent retrievals.
- Lost update problems occur when two transactions read the old value of a variable and then use it to calculate the new value.
- Inconsistent retrieval problems occur when a retrieval transaction runs concurrently with an update transaction.
- Serially equivalent interleaving of two transactions produces the same effect as a serial one.
- Look for conflicting operations.
**read and write Operation Conflict Rules**

- **Conflicting operations**
  - Pair of operations conflicts if their combined effect depends on the order in which they were performed
  - Consider *read* and *write* operations

<table>
<thead>
<tr>
<th>Operations of different transactions</th>
<th>Conflict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>No</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>Yes</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Serial Equivalence and Conflicting Operations

- For two transactions to be *serially equivalent*, it is necessary and sufficient that all pairs of conflicting operations of the two transactions be executed in the same order at all of the objects they both access.

- Consider

  - $T$: $x = \text{read}(i); \text{write}(i, 10); \text{write}(j, 20)$;
  - $U$: $y = \text{read}(j); \text{write}(j, 30); z = \text{read}(i)$;

  Serial equivalence requires that either

  - $T$ accesses $i$ before $U$ and $T$ accesses $j$ before $U$, or
  - $U$ accesses $i$ before $T$ and $U$ accesses $j$ before $T$

- Serial equivalence is used as a criterion for designing concurrency control schemes.
Non-Serially Equivalent Interleaving of $T$ 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></td>
</tr>
</tbody>
</table>

- Each transaction’s access to $i$ and $j$ is serialised with respect to one another, but
- $T$ makes all accesses to $i$ before $U$ does
- $U$ makes all accesses to $j$ before $T$ does
- Therefore this interleaving is not serially equivalent
Concurrency Control Approaches

- Keep in mind we are still talking about a single server
  - May be multi-threaded

- Three alternatives
  - Locking
    - most common, but can lead to deadlock
  - Optimistic concurrency control
    - checks violations at commit time
  - Timestamp ordering
    - checks object timestamps during execution

- To be discussed in more detail
Recoverability from Aborts

- If a transaction aborts, server must make sure other concurrent transactions do not see any of its effects

- Two problems:
  - Dirty reads
    - an interaction between a read operation in one transaction and an earlier write operation on the same object
    - a transaction that committed with a dirty read is not recoverable
  - Premature writes
    - interactions between write operations on the same object by different transactions
    - one of which aborts
## 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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>balance = $a$.getBalance()</th>
<th>$100$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$.setBalance(balance + 10)</td>
<td>$110$</td>
</tr>
</tbody>
</table>

- $a$.setBalance(balance + 10) $110$
- commit transaction

- $U$ has performed a dirty read
- $U$ has committed, so it cannot be undone
Recoverability of Transactions

- If a transaction (like $U$) commits after seeing the effects of a transaction that subsequently aborted, it is not recoverable.
- For recoverability, a commit is delayed until after the commitment of any other transaction whose state has been observed.
  - $U$ waits until $T$ commits or aborts.
  - If $T$ aborts, then $U$ must also abort.
Cascading Aborts

- Suppose that $U$ delays committing until after $T$ aborts
  - Then, $U$ must abort as well
  - If any other transactions have seen the effects due to $U$, they too must be aborted
  - Aborting of these latter transactions may cause still further transactions to be aborted
- Such situations are called \textit{cascading aborts}
- For recoverability, delay the commits
- $U$ could wait to perform \textit{getBalance} until $T$ commits or aborts
  - Allows transaction to proceed without aborting
Avoiding Cascading Aborts

☐ To avoid cascading aborts
   ☐ Transactions are only allowed to read objects written by committed transactions
   ☐ To ensure this, any read operation must be delayed until other transactions that applied a write operation to the same object have committed or aborted

☐ Avoidance of cascading aborts is a stronger condition than recoverability

☐ What impacts does it have?
### Overwriting Uncommitted Values

**Problem:** Interaction between write operations on the same object belonging to different transactions

**Question:** Serially equivalent, but what happens if $U$ aborts?

**Problem of premature writes:**

<table>
<thead>
<tr>
<th>Transaction $T$:</th>
<th>Transaction $U$:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a.setBalance(105)</code></td>
<td><code>a.setBalance(110)</code></td>
</tr>
<tr>
<td>$100$</td>
<td>$110$</td>
</tr>
<tr>
<td><code>a.setBalance(105)</code></td>
<td>$105$</td>
</tr>
</tbody>
</table>

Some database systems keep *before images* and restore them after aborts.
Strict Executions of Transactions

☐ Curing premature writes (with before images)
  ○ Write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted

☐ Strict executions of transactions
  ○ To avoid both dirty reads and premature writes
    ▶ Delay both read and write operations
  ○ Strict executions of transactions
    ▶ Both read and write operations are delayed until all transactions that previously wrote that object have either committed or aborted
  ○ Strict execution of transactions enforces isolation property
Recoverable Server and Tentative Versions

- Server of recoverable objects must be designed so that any updates of objects can be removed if and when a transaction aborts.
- All updates are done on tentative versions of objects:
  - In volatile memory
  - Each transaction has its own private set
- Tentative versions are transferred to the objects only when a transaction commits:
  - Recorded at that time in permanent storage
  - Done in single step without interference
- Tentative versions of objects are deleted when a transaction aborts.
Nested Transactions

Transactions may be composed of other transactions

- Several transactions may be started from within a transaction
- We have a top-level transaction and sub-transactions which may have their own subtransactions
Nested Transactions

- To a parent, a sub-transaction is atomic with respect to failures and concurrent access.
- Transactions at the same level (e.g., $T1$ and $T2$) can run concurrently but access to common objects is serialized.
- A sub-transaction can fail independently of its parent and other sub-transactions.
  - When it aborts, its parent decides what to do:
    - start another subtransaction or give up.
- The CORBA transaction service supports both flat and nested transactions.
Advantages of Nested Transactions (over flat)

- Sub-transactions may run concurrently with other subtransactions at the same level
  - Allows additional concurrency within a transaction
  - Sub-transactions can work in parallel
    - Ex: \textit{branchTotal} operation can be implemented by invoking \textit{getBalance} at every account in the branch

- Sub-transactions can commit or abort independently
  - This is potentially more robust
  - A parent can decide on different actions according to whether a subtransaction has aborted or not
Commitment of Nested Transactions

- A transaction may commit or abort only after its child transactions have completed.
- A sub-transaction decides independently to commit provisionally or to abort.
  - Abort decision is final.
- When a parent aborts, all sub-transactions are aborted.
- When a sub-transaction aborts, the parent can decide whether to abort or not.
- If the top-level transaction commits, then all of the sub-transactions that have provisionally committed can commit too, provided that none of their ancestors has aborted.
Summary on Transactions

- We consider only transactions at a single server
  - Atomic in the presence of concurrent transactions
    - achieved by serially equivalent executions
  - Atomic in the presence of server crashes
    - they save committed state in permanent storage
    - they use strict executions to allow for aborts
    - they use tentative versions to allow for commit/abort
  - Nested transactions structured from sub-transactions
    - they allow concurrent execution of sub-transactions
    - they allow independent recovery of sub-transactions
Introduction to Concurrency Control

- Server execute client’s atomic operations
  - Single, possibly multi-threaded server
- Applicable to servers whose operations can be modelled in terms of Read and Write operations
  - Essential that each operation be atomic
- Protocols designed to cope with conflicts
  - A pair of operations conflicts when their combined effect depends on the order in which they are executed
  - Effect of an operation refers to the value of a data item set by a Write operation and the result returned by a Read operation
Serial Equivalence and Concurrency Control

- Transactions must be scheduled so that their effect on shared objects is serially equivalent

- For serial equivalence
  - All access by a transaction to an object must be serialized with respect to another transaction’s access
  - All pairs of conflicting operations of two transactions should be executed in the same order

- A server can achieve serial equivalence by serialising access to objects (for instance) by the use of locks
  - To ensure, a transaction is not allowed any new locks after it has released a lock

- “Two phase” locking
Concureancy Control Approach

- Servers generally contain large numbers of objects
- A typical transaction access only a few
  - Unlikely to class with other current transactions
  - Can have hot spots
- Locking denies access
- Concurrency control \textit{granularity} is important
  - Determines scope of concurrent access
- Portion of object to which access must be serialized should be as small as possible
- Description of concurrency control schemes does not assume any particular granularity
Concurrency Control Alternatives

- **Locking**
  - A simple example of a serializing mechanism is the use of exclusive locks
  - Server sets a lock with transaction ID on each data item just before it is accessed
  - Removes locks when transaction has completed
  - While locked other transactions must wait or share lock
  - Can lead to deadlock

- **Optimistic concurrency control**
  - Conflicts are checked for before commits

- **Timestamp ordering**
Two-Phase Locking with Exclusive Locks

- Serial equivalence
  - All transaction accesses to data item must be serialized with respect to accesses by other transactions
  - All pairs of conflicting operations should be executed in the same order

- Transaction is not allowed any new locks after it has released a lock
  - “Growing phase”: locks are acquired
  - “Shrinking phase”: locks are released

- Two-phase locking
Transactions T and U with Exclusive Locks

<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(bal*1.1)</td>
<td>b.setBalance(bal*1.1)</td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td>c.withdraw(bal/10)</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>lock B</td>
<td>openTransaction</td>
<td>waits for T's lock on B</td>
</tr>
<tr>
<td>bal = b.getBalance()</td>
<td></td>
<td>bal = b.getBalance()</td>
<td>lock B</td>
</tr>
<tr>
<td>b.setBalance(bal*1.1)</td>
<td>lock A</td>
<td>b.setBalance(bal*1.1)</td>
<td></td>
</tr>
<tr>
<td>a.withdraw(bal/10)</td>
<td>unlock A, B</td>
<td>c.withdraw(bal/10)</td>
<td></td>
</tr>
<tr>
<td>closeTransaction</td>
<td></td>
<td>closeTransaction</td>
<td>unlock B, C</td>
</tr>
</tbody>
</table>

- Initially the balances of A, B and C unlocked
Strict Two-Phase Locking

- Prevent dirty reads and premature writes
  - Transactions needing to read or write data must be delayed until other transactions that wrote the data have committed or aborted
  - Locks must be held during the transaction until it commits or aborts
- **Strict two-phase locking**
- Locks prevent other transactions from reading or writing the data items
- When commit, locks must be held until data items update on permanent storage (ensures recoverability)
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.
Locking Schemes

- Simple exclusive lock used for both read and write operations reduces concurrency more than is necessary.
- Preferable to adopt a locking scheme that controls access to each object to allow:
  - Concurrent transactions reading an object, or
  - Single transaction writing an object
- “Many readers, single writer” scheme
  - Use *read locks* and *write locks*
    - read lock before read operation (shared locks)
    - write lock before write operation
  - Might not be possible to set lock immediately
    - transaction must wait until it is possible to do so
Operation Conflict Rules and Lock Compatibility

- If a transaction T has already performed a read operation on a particular object, then a concurrent transaction U must not write that object until T commits or aborts.

- If a transaction T has already performed a write operation on a particular object, then a concurrent transaction U must not read or write that object until T commits or aborts.

<table>
<thead>
<tr>
<th>For one object</th>
<th>Lock requested</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read</td>
</tr>
<tr>
<td>Lock already set</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>read</td>
</tr>
<tr>
<td></td>
<td>write</td>
</tr>
</tbody>
</table>
Inconsistent Retrievals / Lost Updates

☐ Inconsistent retrievals and lost updates are caused by conflicts between Read and Write operations

☐ Inconsistent retrievals

☑ Prevent by performing retrieval transaction before or after the update transactions
  ➢ retrieval first: read locks delay the update
  ➢ retrieval second: read locks delayed until update

☐ Lost updates

☑ Two transactions read a data item and use value to calculate a new value

☑ Prevent by having later transactions delay reads
Lock Promotion

- Read locks are promoted to write locks when read data is to be written
  - Delays subsequent transaction requiring a read lock
- A transaction with a shared read lock cannot promote its read lock to a write lock because it would conflict with the read locks
  - Must wait for the read locks to be released
- Lock promotion refers to the conversion of a lock to a stronger lock (i.e., more exclusive)
- It is not safe to demote a lock held before commit
Lock Implementation

- A lock manager in the server grants locks
- Responsible for maintaining table of locks for data
- Entry description:
  - Transaction ID’s (multiple for shared locks)
  - Data item ID
  - Lock type
  - Condition variable
- Each client runs in a separate server thread
  - Waits on condition variable when lock not available
  - Condition variable is signalled on unlock
The granting of locks will be implemented by the lock manager in the server
- Holds a set of locks
- Each lock is an instance of the *class Lock*
- Variables refer to the object
  - holder(s) of the lock and its type

Lock manager code uses wait (when an object is locked) and notify when the lock is released

Lock manager provides *setLock* and *unLock* operations for use by the server
public class Lock {
    private Object object; // the object being protected by the lock
    private Vector holders; // the TIDs of current holders
    private LockType lockType; // the current type
    public synchronized void acquire(TransID trans, LockType aLockType) {
        while (/*another transaction holds the lock in conflicting mode*/) {
            try {
                wait();
            } catch (InterruptedException e) {/*...*/ }
        }
        if (holders.isEmpty()) { // no TIDs hold lock
            holders.addElement(trans);
            lockType = aLockType;
        } else if (/*another transaction holds the lock, share it*/) {
            if (/*this transaction not a holder*/) {
                holders.addElement(trans);
            } else if (/*this transaction is a holder but needs a more exclusive lock*/) {
                lockType.promote();
            }
        }
    }
}
Lock Class (continued)

```java
public synchronized void release(TransID trans ){
    holders.removeElement(trans); // remove this holder
    // set locktype to none
    notifyAll();
}
```
public class LockManager {
    private Hashtable theLocks;

    public void setLock(Object object, TransID trans, LockType lockType) {
        Lock foundLock;
        synchronized(this) {
            // find the lock associated with object
            // if there isn’t one, create it and add to the hashtable
        }
        foundLock.acquire(trans, lockType);
    }

    // synchronize this one because we want to remove all entries
    public synchronized void unLock(TransID trans) {
        Enumeration e = theLocks.elements();
        while (e.hasMoreElements()) {
            Lock aLock = (Lock)(e.nextElement());
            if (/* trans is a holder of this lock */ ) aLock.release(trans);
        }
    }
}
Deadlocks

- Use of locks can lead to deadlock
  - Two transactions are waiting and each depends on the other to release a lock so it can resume
- Deadlocks less likely to occur if operations were to request write locks initially
- Formally, deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock
- Necessarily involves write locks
- Shared read locks do not deadlock
# 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></td>
<td>$a.deposit(100)$;</td>
<td>write lock $A$</td>
<td></td>
<td>$b.deposit(200)$</td>
<td>write lock $B$</td>
</tr>
<tr>
<td></td>
<td>$b.withdraw(100)$</td>
<td>waits for $U$’s lock on $B$</td>
<td></td>
<td></td>
<td>waits for $T$’s lock on $A$</td>
</tr>
<tr>
<td></td>
<td>$\cdots$</td>
<td></td>
<td></td>
<td>$a.withdraw(200)$;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\cdots$</td>
<td></td>
<td></td>
<td>$\cdots$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\cdots$</td>
<td></td>
<td></td>
<td>$\cdots$</td>
<td></td>
</tr>
</tbody>
</table>

- The *deposit* and *withdraw* methods are atomic
  - They read as well as write
  - They acquire write locks
Deadlocks and Wait-for Graphs

- Deadlocks can be represented using \textit{wait-for} graph
  - Represents waiting relationships between current transactions at a server
  - Nodes represent transactions
  - Edge represent wait-for relationships among transactions
  - Edge from node T to node U when transaction T is waiting for transaction U to release a lock
- If one of the transactions in a cycle is aborted, then its locks are released and deadlocks broken
- Although each transaction can wait for only one data item at a time, it may be involved in several cycles
Definition of deadlock

- Deadlock is a state in which each member of a group of transactions is waiting for some other member to release a lock.

- A wait-for graph can be used to represent the waiting relationships between current transactions.
A Cycle in a Wait-For Graph

- Wait-for graph cycle: T → U → ... → V → T
  - Each transaction waits for the next in the cycle
  - All transactions are blocked waiting for locks
  - None of the locks can ever be released (deadlocked)
  - If one transaction is aborted, then its locks are released
  - Cycle is broken
Another Wait-For Graph

- T, U, and V share a read lock on C
- W holds write lock on B (which V is waiting for)
- T and W then request write locks on C
- Deadlock occurs (Why?)
Deadlock Prevention is Unrealistic

☐ What if we lock all objects used by a transaction
  ☒ When it starts
  ☒ Unnecessarily restricts access to shared resources
  ☒ Might be impossible to predict at the start of a transaction which objects will be used

☐ Suppose we requesting locks in some predefined order
  ☒ Can result in premature locking
  ☒ Can lead to reduction in concurrency

☐ Trade off of allowing the possibility of deadlocks and greater concurrency and performance
Deadlock Detection

- How do we know deadlocks have occurred
  - By finding cycles in the wait-for graph (detection)
  - Transaction must be selected to abort (breaks cycle)
  - Deadlock detection can be part of the lock manager
  - Use wait-for graph to check for cycles (when?)
  - Edges added / removed by the lock manager
    - setLock and unLock operations
  - When a cycle is detected, choose a transaction to abort
    - Remove all the edges belonging to it from the graph
  - Hard to choose a victim
    - What happens if we choose the oldest transaction?
Deadlock Prevention

- One solution is to prevent deadlocks
- Simple way
  - Lock all the data items used when transaction starts
  - Unnecessarily restricts access to shared resources
  - Sometimes impossible to predict data items used
- Another way
  - Requesting locks on data items in a predefined order
  - Can result in premature locking
  - Can result in reduction in concurrency
Deadlock Detection

- Server find cycles in the wait-for graph
- Server selects a transaction to abort to break cycle
- Software responsible for deadlock detection can be part of the lock manager
  - Edges added by the lock operations
    - an edge $T \rightarrow U$ is added whenever the lock manager blocks a request by $T$ for a lock on data already locked by $U$
  - Edges removed by the unlock operations
    - an edge $T \rightarrow U$ is deleted whenever $U$ releases a lock $T$ is waiting for and allows $T$ to proceed
Timeouts on Locks

- Lock timeouts can be used to resolve deadlocks
  - Each lock is invulnerable for a given a limited period
  - After this time, a lock becomes vulnerable
  - Vulnerable lock remains if no other transaction waiting
  - If any other transaction is waiting to access the object protected by a vulnerable lock, the lock is broken
    - Object is unlocked and the waiting transaction resumes
  - Transaction whose lock broken is normally aborted

- Problems?
  - Break locks without deadlocks
  - Long transactions are penalized
  - Hard to set timeout values
## Resolution of the 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><em>a.deposit(100)</em>;</td>
<td>write lock A</td>
<td></td>
<td><em>b.deposit(200)</em>;</td>
<td>write lock B</td>
</tr>
<tr>
<td></td>
<td><em>b.withdraw(100)</em></td>
<td><em>waits for U’s lock on B</em> (timeout elapses)</td>
<td></td>
<td><em>a.withdraw(200)</em>;</td>
<td><em>waits for T’s lock on A</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>T’s lock on A becomes vulnerable,</em> unlock A, abort T</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[\text{\ldots}\]
Increasing Concurrency in Locking Schemes

- Two-version locking
  - Allows one transaction to write tentative versions of data items while other transactions read from the committed versions
  - Read operations only wait if another transaction is currently committing the same data item
  - Allows more concurrency than read-write locks
  - Writing transactions risk waiting or even rejection when attempt to commit
    - wait until reading transactions have completed
    - deadlocks may occur when transactions are waiting
XDFS File Server with Two-Version Locking

- Three types of locks:
  - *read* lock, *write* lock, and *commit* lock
- Read lock on data item before Read operation
  - Waits only if data item has a *commit* lock
- Write lock on data item before Write operation
  - Waits if data item has a *write* lock or *commit* lock
- Commit request
  - Attempt to convert *write* locks to *commit* locks
  - Wait for *read* locks to be released
Hierarchical Locks

- Use hierarchy of locks with different granularities

- At each level, setting of parent lock has the same effect as setting all equivalent child locks
  - each node in the hierarchy can be locked
    - explicit access to the node
    - implicit access to its children

- Child node read/write lock
  - intention to read/write lock on parent and ancestors
    - compatible with other intention locks
    - conflicts with read and write locks
Optimistic Concurrency Control

- Drawbacks of locking
  - Overhead of lock maintenance and need to use locking even for read-only transactions
  - Use of locks can result in deadlock and prevention or detection and recovery reduces concurrency
  - To avoid cascading aborts, locks cannot be released until the end of the transaction

- Optimistic approach
  - Transactions proceed as if no conflict will occur
  - Conflicts cause transactions to be aborted/restarted
Optimistic Concurrency Control (continued)

- Optimistic because likelihood of conflict is low
- Transaction proceeds without restriction until the `closeTransaction` (no waiting, therefore no deadlock)
- Then checked to see whether it has a conflict
  - When a conflict arises, a transaction is aborted
- Each transaction has three phases:
  - *Working phase* - uses tentative versions of objects, recording read and write sets for each transaction
  - *Validation phase* - look for conflicts at close, validate or resolve conflict with choice of which to abort
  - *Update phase* - tentative versions are made permanent, read only can commit immediately
Optimistic Transaction Phases

- Transaction phases
  - Read phase
    - each transaction has a tentative version of data
    - read operations performed immediately (committed)
    - write operations record new values as tentative values
    - two records
      - *read set*: data items read by transactions
      - *write set*: data items written by transactions
  - Validation phase (*CloseTransaction* request)
    - transaction validated to determine data conflicts
    - if successful, transactions can commit
Optimistic Transaction Phases (continued)

- if unsuccessful, conflict resolution enforced

- Write phase
  - if transaction validated, all changes recorded in its tentative versions are made permanent
Validation of Transactions

- To assist in performing validation, each transaction is assigned a transaction number when it enters the validation phase
  - Retained if successful or released if failure
- Monotonically increasing transaction numbers
  - Transaction number defines position in time
  - A transaction always finishes its read phase after all transactions with lower numbers $j$
- Validation test is based on conflicts between operations in pairs of transactions $T_i$ and $T_v$ (transaction being validated)
  - Ensures serializability
Validation of Transactions (continued)

- $T_i / T_v$
  - Read/Write: $T_i$ must not read items written by $T_v$
  - Write/Read: $T_v$ must not read items written by $T_i$
  - Write/Write: $T_i$ must not write items written by $T_v$ and $T_v$ must not write item written by $T_i$

- Make simplification that only one transaction may be in the validation and write phase at once

- Validation must test for overlaps

- Two forms of validation
  - *Backward validation*
  - *Forward validation*
Validation of Transactions

Earlier committed transactions

Working Validation Update

Transaction being validated

Earlier committed transactions

Later active transactions

active₁ active₂
Backward Validation

- Check $T_v$ with preceding overlapping transactions
  - $T_v$ reads not affected by preceding reads
  - Check if read set overlaps with any preceding overlapping write sets
- Only way to resolve any conflicts is to abort the transaction undergoing validation
- Write sets of old committed versions of data items corresponding to recently committed transactions are retained until there are no unvalidated overlapping transactions with which they might conflict
Backward Validation of Transactions

Backward validation of transaction $T_v$

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

- $startT_n$ is the biggest transaction number assigned to some other committed transaction when $T_v$ started its working phase
- $finishT_n$ is biggest transaction number assigned to some other committed transaction when $T_v$ started its validation phase
- $StartT_n + 1 = T_2$ and $finishT_n = T_3$
  - Read set of $T_v$ must be compared with write sets of $T_2$ and $T_3$
- The only way to resolve a conflict is to abort $T_v$
Forward Validation

- Write set of $T_v$ compared with the read sets of all overlapping active transactions, those still in their read phase

- Alternatives to resolve conflicts
  - Defer validation until conflicting transactions have finished, but further conflicting ones may start
  - Abort all conflicting active transactions and commit the transaction being validated
  - Abort transaction being validated
Forward Validation

Forward validation of transaction $T_v$

```java
boolean valid = true;
for (int $T_id = active_1; T_id <= active_N; T_id++){
    if (write set of $T_v$ intersects read set of $T_{id}$) valid = false;
}
```

- Read sets of active transactions may change during validation
- Rule 1. the write set of $T_v$ is compared with the read sets of all overlapping active transactions
  - Compare write set of $T_v$ with read sets of active$_1$ and active$_2$
- Rule 2. (read $T_v$ vs write $T_i$) is automatically fulfilled because the active transactions do not write until after $T_v$ has completed
Backward vs. Forward Validation

☐ Forward validation allows flexibility in resolution of conflicts where backward has only one choice

☐ Read sets of transactions are larger than write sets
   ☐ Backward validation checks possibly large read set with old write sets
   ☐ Forward validation checks small write set with read sets of active transactions

☐ Backward validation stores old write sets

☐ Forward validation has to allow for new transactions

☐ Starvation
   ☐ After a transaction is aborted, the client must restart it, but there is no guarantee it will ever succeed
Timestamp Ordering Concurrency Control

- Each operation in a transaction validated when carried out
  - If an operation cannot be validated, the transaction is aborted
  - Each transaction is given a unique timestamp when it starts
    - Defines its position in the time sequence of transactions
  - Transaction requests can be totally ordered by their timestamps

- Basic timestamp ordering rule (based on operation conflicts)
  - A request to write an object is valid only if that object was last read and written by earlier transactions
  - A request to read an object is valid only if that object was last written by an earlier transaction

- This rule assumes only one version of each object
- Refine the rule to make use of the tentative versions
  - To allow concurrent access by transactions to objects
Operation Conflicts for Timestamp Ordering

- **Refined rule**
  - Tentative versions are committed in order of their timestamps (may wait) but no need for client to wait
  - Read operations wait for earlier transactions to finish
    - only wait for earlier ones (no deadlock)
  - Each read/write operation is checked with conflict rules

<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>
</tbody>
</table>
|      | $T_c$ must not write an object that has been read by any $T_i$ where $T_i > T_c$
|      | this requires that $T_c \geq$ the maximum read timestamp of the object. |
| 2.   | write   | write   |
|      | $T_c$ must not write an object that has been written by any $T_i$ where $T_i > T_c$
|      | this requires that $T_c >$ write timestamp of the committed object. |
| 3.   | read    | write   |
|      | $T_c$ must not read an object that has been written by any $T_i$ where $T_i > T_c$
|      | this requires that $T_c >$ write timestamp of the committed object. |
Write Operations and Timestamps

- Versions and timestamps when we do $T_3$ write
- To be allowed, $T_3 \geq$ maximum read timestamp

(a) $T_3$ write

Before

After

T2
T3

Time

(b) $T_3$ write

Before

After

T1
T2
T3

Time

(c) $T_3$ write

Before

After

T1
T4
T1
T3
T4

Time

(d) $T_3$ write

Before

After

T4

Transaction aborts

Time

Key:

- Committed
- Tentative

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

$T_1 < T_2 < T_3 < T_4$
Timestamp Ordering Write Rule

- By combining rules 1 (write/read) and 2 (write/write) we have the following rule for deciding whether to accept a write operation requested by transaction $T_c$ on object D
  - Rule 3 does not apply to writes

```plaintext
if ($T_c \geq \text{maximum read timestamp on } D \land \land$
  $T_c > \text{write timestamp on committed version of } D)$
  perform write operation on tentative version of D with write timestamp $T_c$
else /* write is too late */
  Abort transaction $T_c$
```
Using Rule 3 we get the following rule for deciding what to do about a read operation requested by transaction $T_c$ on object $D$

- accept it immediately, wait, or reject it

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

- Illustrates the timestamp, ordering read rule, in each case we have T3 read
- A version whose write timestamp is <= T3 is selected

(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 $T_i$)

$T_1 < T_2 < T_3 < T_4$
Transaction Commits with Timestamp Ordering

- When a coordinator receives a commit request, it will always be able to carry it out because all operations have been checked for consistency with earlier transactions
  - Committed versions of an object must be created in timestamp order
  - Server may sometimes need to wait, but the client need not wait
  - To ensure recoverability, the server will save the “waiting to be committed versions” in permanent storage

- Timestamp ordering algorithm is strict because
  - The read rule delays each read operation until previous transactions that had written the object had committed or aborted
  - Writing the committed versions in order ensures that the write operation is delayed until previous transactions that had written the object have committed or aborted
Timestamp Ordering Concurrency Control

- Method avoids deadlocks
- But is likely to suffer from restarts
- *Ignore obsolete write*
  - If a write is too late it can be ignored instead of aborting the transaction, because if it had arrived in time its effects would have been overwritten anyway
  - However, if another transaction has read the object, the transaction with the late write fails due to the read timestamp on the item
- *Multiversion timestamp ordering*
  - Allows more concurrency by keeping multiple committed versions
  - Late read operations need not be aborted
## Timestamps in Transactions $T$ and $U$

<table>
<thead>
<tr>
<th>$T$</th>
<th>$U$</th>
<th>Timestamps and versions of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>openTransaction</td>
<td>b.setBalance(bal*1.1)</td>
<td>${T}$</td>
</tr>
<tr>
<td>bal = b.getBalance()</td>
<td>b.setBalance(bal*1.1)</td>
<td>$S, T$</td>
</tr>
<tr>
<td>$\bullet \bullet \bullet$</td>
<td>wait for $T$</td>
<td>$S, T$</td>
</tr>
<tr>
<td>commit</td>
<td>a.withdraw(bal/10)</td>
<td>$T$</td>
</tr>
<tr>
<td>bal = b.getBalance()</td>
<td>c.withdraw(bal/10)</td>
<td>$T, U$</td>
</tr>
<tr>
<td>bal = b.getBalance()</td>
<td>b.setBalance(bal*1.1)</td>
<td>${U}$</td>
</tr>
<tr>
<td>commit</td>
<td>c.setBalance(bal*1.1)</td>
<td>$S, U$</td>
</tr>
</tbody>
</table>
Late write Operation Would Invalidate a read

Key:
- Committed: $T_i$, $T_k$
- Tentative: $T_i$, $T_k$

Object produced by transaction $T_i$ (with write timestamp $T_i$ and read timestamp $T_k$)

$T_1 < T_2 < T_3 < T_4 < T_5$

$T_3$ read; $T_3$ write; $T_5$ read; $T_4$ write;
Comparing Methods for Concurrency Control

- Pessimistic approach (detect conflicts as they arise)
  - Timestamp ordering
    - serialisation order decided statically
    - better for transactions where reads >> writes
    - strategy for aborts - immediate
  - Locking
    - serialisation order decided dynamically
    - better for transactions where writes >> reads
    - strategy for aborts - waits but can get deadlock

- Optimistic methods
  - All transactions proceed, but may need to abort at the end
  - Efficient operations when few conflicts
  - Aborts lead to repeating work

- Above methods are not always adequate
  - In cooperative work may need user notification and involvement
Summary

- Operation conflicts form a basis for the derivation of concurrency control protocols
  - Protocols ensure serializability and allow for recovery by using strict executions (e.g., to avoid cascading aborts)
- Three alternative strategies are possible in scheduling an operation in a transaction:
  1. to execute it immediately, (2) to delay it, or (3) to abort it
  2. Strict two-phase locking uses (1) and (2)
     - aborting in the case of deadlock
     - ordering according to when transactions access common objects
  3. Timestamp ordering uses all three - no deadlocks
     - ordering according to the time transactions start
  4. Optimistic concurrency control allows transactions to proceed without any form of checking until they are completed.
     - validation is carried out and starvation can occur