Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal

Acknowledgements for the materials: Indy Gupta and Nikita Borisov
Logistics

• Midterm 2 in two weeks (April 2-4).

• Same format and duration as Midterm 1.

• Syllabus: Mutual Exclusion, Leader Election, Synchronous Consensus, Paxos, Raft.

• You should be able to make reservations on PrairieTest starting tomorrow (March 21\textsuperscript{st}).

• Some practice questions for Midterm 2 will be released next week.
Agenda for today

- Transaction Processing and Concurrency Control
  - Chapter 16
    - Transaction semantics: ACID
    - Isolation and serial equivalence
    - Conflicting operations
    - Two-phase locking
    - Deadlocks
    - Timestamped ordering
- Distributed Transactions (if time)
Transaction Properties: ACID

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

*How to prevent transactions from affecting each other?*

- Execute them serially at the server (one at a time).
  - e.g. through a global lock.
  - But this reduces number of concurrent transactions

*Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).*
Concurrent Control: Two approaches

• **Pessimistic**: assume the worst, prevent transactions from accessing the same object
  • E.g., Locking

• **Optimistic**: assume the best, allow transactions to write, but check later
  • E.g., Check at commit time
Pessimistic: Locking

• Grabbing a global lock is wasteful
  • what if no two transactions access the same object?

• Each object has a lock
  • can further improve concurrency.
  • reads on the same object are non-conflicting.

• Per-object read-write locks.
  • Read mode: multiple transactions allowed in
  • Write mode: exclusive lock
Guaranteeing Serial Equivalence with Locks

- **Two-phase locking**
  - A transaction cannot acquire (or promote) any locks after it has started releasing locks
  - Transaction has two phases
    1. Growing phase: only acquires or promotes locks
    2. Shrinking phase: only releases locks
      - **Strict two phase locking**: releases locks only at commit point
## Can lead to Deadlocks!

### Transaction T1
- `read_lock(x)`
- `x = getSeats(ABC123);`
- `if(x > 1)`
  - `x = x - 1;`
- `write_lock(x) Blocked!`
- `write(x, ABC123);`
- `commit`

### Transaction T2
- `read_lock(x)`
- `x = getSeats(ABC123);`
- `if(x > 1)`
  - `x = x - 1;`
- `write_lock(x) Blocked!`
- `write(x, ABC123);`
- `commit`

---

**Deadlock!**
When do deadlocks occur?

• 3 necessary conditions for a deadlock to occur
  1. Some objects are accessed in exclusive lock modes
  2. Transactions holding locks are not preempted
  3. There is a circular wait (cycle) in the Wait-for graph

• “Necessary” = if there’s a deadlock, these conditions are all definitely true

• (Conditions not sufficient: if they’re present, it doesn’t imply a deadlock is present.)
Combating Deadlocks

1. Lock all objects in the beginning in a single atomic step.
   • no circular wait-for graph created (3rd deadlock condition breaks)
     - may not know of all operations a priori.

2. Lock timeout: abort transaction if lock cannot be acquired within timeout
   • (2nd deadlock condition breaks)
     - Expensive; leads to wasted work
     - How to determine the timeout value?
       • Too large: long delays
       • Too small: false positives.

3. Deadlock Detection:
   • keep track of Wait-for graph, and find cycles in it (e.g., periodically)
   • If find cycle, there’s a deadlock
     ⇒ Abort one or more transactions to break cycle (2nd deadlock condition breaks)
Concurrency Control: Two approaches

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking

- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time
Optimistic Concurrency Control

• Increases concurrency more than pessimistic concurrency control
• Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon’s Dynamo
• Preferable than pessimistic when conflicts are expected to be rare
  • But still need to ensure conflicts are caught!
First cut approach

- Most basic approach
  - Write and read objects at will
  - Check for serial equivalence at commit time
  - If abort, roll back updates made
  - An abort may result in other transactions that read dirty data, also being aborted
    - Any transactions that read from those transactions also now need to be aborted
      😞 Cascading aborts
Timestamped ordering

• Assign each transaction an id
• Transaction id determines its position in serialization order.
• Ensure that for a transaction T, both are true:
  1. T’s write to object O allowed only if transactions that have read or written O had lower ids than T.
  2. T’s read to object O is allowed only if O was last written by a transaction with a lower id than T.
• Implemented by maintaining read and write timestamps for the object
• If rule violated, abort!
• Never results in a deadlock! Older transaction never waits on newer ones.
Timestamped ordering: per-object state

- Committed value.
- Transaction id (timestamp) that wrote the committed value.
- Read timestamps (RTS): List of transaction ids (timestamps) that have read the committed value.
- Tentative writes (TW): List of tentative writes sorted by the corresponding transaction ids (timestamps).
  - Timestamped versions of the object.
## Timestamped ordering rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>$T_c$</th>
<th>$T_i$</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. write</td>
<td>read</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>This requires that $T_c \geq$ the maximum read timestamp of the object.</td>
</tr>
<tr>
<td>2. write</td>
<td>write</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>This requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
<tr>
<td>3. read</td>
<td>write</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>This requires that $T_c &gt;$ write timestamp of the committed object.</td>
</tr>
</tbody>
</table>
Timestamped ordering: write rule

Transaction $T_c$ requests a write operation on object $D$
if ($T_c \geq$ max. read timestamp on $D$
    && $T_c >$ write timestamp on committed version of $D$)
    
    Perform a tentative write on $D$:
    
    If $T_c$ already has an entry in the TW list for $D$, update it.
    Else, add $T_c$ and its write value to the TW list.

else

    abort transaction $T_c$

    // too late; a transaction with later timestamp has already read or written the object.
Timestamped ordering: write rule

(a) \( T_3 \) write

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_2 )</td>
<td>( T_2 ) ( T_3 )</td>
</tr>
</tbody>
</table>

Time

(b) \( T_3 \) write

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 ) ( T_2 )</td>
<td>( T_1 ) ( T_2 ) ( T_3 )</td>
</tr>
</tbody>
</table>

Time

(c) \( T_3 \) write

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 ) ( T_4 )</td>
<td>( T_1 ) ( T_3 ) ( T_4 )</td>
</tr>
</tbody>
</table>

Time

(d) \( T_3 \) write

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_4 )</td>
<td>( T_4 )</td>
</tr>
</tbody>
</table>

Time

Key:
- \( T_i \): Committed
- \( T_i \): Tentative

\( T_1 < T_2 < T_3 < T_4 \)

Read timestamps not shown in this example.
(assume zero reads)
Timestamped ordering: read rule

Transaction $T_c$ requests a read operation on object $D$

if ($T_c > \text{write timestamp on committed version of } D$) {
  $D_s = \text{version of } D \text{ with the maximum write timestamp that is } \leq T_c$
  //search across the committed timestamp and the TW list for object $D$.
  if ($D_s$ is committed)
    read $D_s$ and add $T_c$ to RTS list (if not already added)
  else
    if $D_s$ was written by $T_c$, simply read $D_s$
    else
      wait until the transaction that wrote $D_s$ is committed or aborted, and reapply the read rule.
      // if the transaction is committed, $T_c$ will read its value after the wait.
      // if the transaction is aborted, $T_c$ will read the value from an older transaction.
  }
else
  abort transaction $T_c$
  //too late; a transaction with later timestamp has already written the object.
Timestamped ordering: read rule

(a) T₃ read

(b) T₃ read

(c) T₃ read

(d) T₃ read

Key:

Committed

Tentative

T₁ < T₂ < T₃ < T₄
Timestamped ordering: committing

- Suppose $T_4$ is ready to commit.
- Must wait until $T_3$ commits or aborts.

- When a transaction is committed, the committed value of the object and associated timestamp are updated, and the corresponding write is removed from TW list.
Lost Update Example with Timestamped Ordering

**Transaction T1**
\[ x = \text{getSeats}(ABC123); \]
\[ \text{if}(x > 1) \]
\[ x = x - 1; \]
\[ \text{write}(x, \text{ABC123}); \]
\[ \text{commit} \]

**Transaction T2**
\[ x = \text{getSeats}(ABC123); \]
\[ \text{if}(x > 1) \]
\[ x = x - 1; \]
\[ \text{write}(x, \text{ABC123}); \]
\[ \text{commit} \]

---

ABC123: state committed value = 10 committed timestamp = 0
RTS:
TW:
Lost Update Example with Timestamped Ordering

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = \text{getSeats}(ABC123); )</td>
<td>( x = \text{getSeats}(ABC123); )</td>
</tr>
<tr>
<td>( \text{if}(x &gt; 1) ) ( x = x - 1; )</td>
<td>( \text{if}(x &gt; 1) ) ( x = x - 1; )</td>
</tr>
<tr>
<td>\text{write}(x, ABC123);</td>
<td>\text{write}(x, ABC123);</td>
</tr>
<tr>
<td>\text{commit}</td>
<td>\text{commit}</td>
</tr>
</tbody>
</table>

ABC123: state
- committed value = 10
- committed timestamp = 0

RTS: 1
TW:
Lost Update Example with Timestamped Ordering

**Transaction T1**

\[
x = \text{getSeats}(ABC123);
\]

\[
\text{if}(x > 1) \quad x = x - 1;
\]

write\((x, ABC123)\);

commit

**Transaction T2**

\[
x = \text{getSeats}(ABC123);
\]

\[
\text{if}(x > 1) \quad x = x - 1;
\]

write\((x, ABC123)\);

commit

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1
TW:
Lost Update Example with Timestamped Ordering

Transaction T1
\[
\begin{align*}
x &= \text{getSeats}(\text{ABC123}); \\
\text{if}(x > 1) &\quad \text{if}(x > 1) \\
  x &= x - 1; \\
\text{write}(x, \text{ABC123}); &\quad \text{write}(x, \text{ABC123}); \\
\text{commit} &\quad \text{commit}
\end{align*}
\]

Transaction T2
\[
\begin{align*}
x &= \text{getSeats}(\text{ABC123}); \\
\text{if}(x > 1) &\quad \text{if}(x > 1) \\
  x &= x - 1; \\
\text{write}(x, \text{ABC123}); &\quad \text{write}(x, \text{ABC123}); \\
\text{commit} &\quad \text{commit}
\end{align*}
\]

ABC123: state
committed value = 10
committed timestamp = 0
RTS: 1, 2
TW:
Lost Update Example with Timestamped Ordering

**Transaction T1**

\[ x = \text{getSeats}(ABC123); \]
\[ \text{if}(x > 1) \]
\[ x = x - 1; \]
\[ \text{write}(x, ABC123); \]
\[ \text{commit} \]

**Transaction T2**

\[ x = \text{getSeats}(ABC123); \]
\[ \text{if}(x > 1) \]
\[ x = x - 1; \]
\[ \text{write}(x, ABC123); \]
\[ \text{commit} \]

ABC123: state committed value = 10 committed timestamp = 0
RTS: 1, 2
TW:
Abort!
Next Example with Timestamped Ordering

**Transaction T1**

\[ x = \text{getSeats}(ABC123); \]
\[ y = \text{getSeats}(ABC789); \]
\[ \text{write}(x-5, \text{ABC123}); \]
\[ \text{write}(y+5, \text{ABC789}); \]
\[ \text{commit} \]

**Transaction T2**

\[ x = \text{getSeats}(ABC123); \]
\[ y = \text{getSeats}(ABC789); \]
\[ \text{print}(\text{“Total:”} \ x+y); \]
\[ \text{commit} \]

ABC123: state
committed value = 10
committed timestamp = 0
RTS:
TW:

ABC789: state
committed value = 5
committed timestamp = 0
RTS:
TW:

Try on your own! Will discuss in next class.
Timestamped ordering

- Assign each transaction an id
- Transaction id determines its position in serialization order.
- Ensure that for a transaction T, both are true:
  1. T’s write to object O allowed only if transactions that have read or written O had lower ids than T.
  2. T’s read to object O is allowed only if O was last written by a transaction with a lower id than T.
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
- Never results in a deadlock! Older transaction never waits on newer ones.
Concurrency Control: Summary

• How to prevent transactions from affecting one another?
• Goal: increase concurrency and transaction throughput while maintaining correctness (ACID).
• Target serial equivalence.
• Two approaches:
  • Pessimistic concurrency control: locking based.
    • read-write locks with two-phase locking and deadlock detection.
  • Optimistic concurrency control: abort if too late.
    • timestamped ordering.