Distributed Databases

CS347
Lecture 15
June 4, 2001

Topics for the day

- Concurrency Control
  - Schedules and Serializability
  - Locking
  - Timestamp control
- Reliability
  - Failure models
  - Two-phase commit protocol

Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
</tbody>
</table>

Node 1, Node 2

Constraint: X = Y

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a ← X</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X ← a+100</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>b ← Y</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Y ← b+100</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>c ← X</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>X ← 2c</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>d ← Y</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Y ← 2d</td>
<td></td>
</tr>
</tbody>
</table>

Possible Schedule

(node X)

1. \( (T_1) \ a ← X \)
2. \( (T_1) \ X ← a+100 \)
5. \( (T_2) \ c ← X \)
6. \( (T_2) \ X ← 2c \)

(node Y)

3. \( (T_1) \ b ← Y \)
4. \( (T_1) \ Y ← b+100 \)
7. \( (T_2) \ d ← Y \)
8. \( (T_2) \ Y ← 2d \)

If \( X = Y = 0 \) initially, \( X = Y = 200 \) at end

Precedence: intra-transaction
Inter-transaction
Definition of a Schedule

Let \( T = \{T_1, T_2, \ldots, T_N\} \) be a set of transactions. A schedule \( S \) over \( T \) is a partial order with ordering relation \( \prec_S \) where:

1. \( S = \bigcup T_i \)
2. \( \prec_S \supseteq \bigcup \prec_i \)
3. for any two conflicting operations \( p, q \in S \), either \( p \prec_S q \) or \( q \prec_S p \)

Note: In centralized systems, we assumed \( S \) was a total order and so condition (3) was unnecessary.

Example

\[
\begin{align*}
(T_1) & \quad r_1[X] \rightarrow w_1[X] \\
(T_2) & \quad r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X] \\
(T_3) & \quad r_3[X] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z] \\
\end{align*}
\]

\[
\begin{align*}
S: & \quad r_3[Y] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z] \\
& \quad r_1[X] \rightarrow w_1[X] \\
& \quad \text{P(S): } T_2 \rightarrow T_1 \rightarrow T_3
\end{align*}
\]

Precedence Graph

- Precedence graph \( P(S) \) for schedule \( S \) is a directed graph where
  - Nodes = \( \{T_i \mid T_i \text{ occurs in } S\} \)
  - Edges = \( \{T_i \rightarrow T_j \mid \exists p \in T_i, q \in T_j \text{ such that } p, q \text{ conflict and } p \prec_S q\} \)

\[
\begin{align*}
T_3 & \rightarrow T_1 \rightarrow T_2 \\
r_3[X] & \rightarrow w_3[X] \\
r_1[X] & \rightarrow w_1[X] \rightarrow w_1[Y] \\
r_3[X] & \rightarrow w_3[Y]
\end{align*}
\]

Serializability

**Theorem:** A schedule \( S \) is serializable iff \( P(S) \) is acyclic.

**Enforcing Serializability**
- Locking
- Timestamp control
**Distributed Locking**
- Each lock manager maintains locks for local database elements.
- A transaction interacts with multiple lock managers.

![Diagram of lock managers and transactions](image)

**Locking Rules**
- Well-formed/consistent transactions
  - Each transaction gets and releases locks appropriately
- Legal schedulers
  - Schedulers enforce lock semantics
- Two-phase locking
  - In every transaction, all lock requests precede all unlock requests.

These rules guarantee serializable schedules

---

**Locking replicated elements**
- Example:
  - Element X replicated as X₁ and X₂ on sites 1 and 2
  - T obtains read lock on X₁; U obtains write lock on X₂
  - Possible for X₁ and X₂ values to diverge
  - Possible that schedule may be unserializable
- How do we get global lock on logical element X from local locks on one or more copies of X?

**Primary-Copy Locking**
- For each element Xᵢ designate specific copy Xᵢ as primary copy
- Local-lock(Xᵢ) ⇒ Global-lock(X)

**Synthesizing Global Locks**
- Element X with n copies X₁ .... Xₙ
- Choose “s” and “x” such that
  - 2x > n
  - s + x > n
- Shared-lock(s copies) ⇒ Global-shared-lock(X)
- Exclusive-lock(x copies) ⇒ Global-exclusive-lock(X)
Special cases

**Read-Lock-One; Write-Locks-All** \((s = 1, x = n)\)
- Global shared locks inexpensive
- Global exclusive locks very expensive
- Useful when most transactions are read-only

**Majority Locking** \((s = x = \lceil (n+1)/2 \rceil)\)
- Many messages for both kinds of locks
- Acceptable for broadcast environments
- Partial operation under disconnected network possible

---

**Timestamp Ordering Schedulers**

**Basic idea:** Assign timestamp \(ts(T)\) to transaction \(T\).
If \(ts(T_1) < ts(T_2) < \ldots < ts(T_n)\), then scheduler produces schedule equivalent to serial schedule \(T_1 T_2 T_3 \ldots T_n\).

**TO Rule:** If \(p_i[X]\) and \(q_j[X]\) are conflicting operations, then \(p_i[X] <_S q_j[X]\) iff \(ts(T_i) < ts(T_j)\).

**Theorem:** If \(S\) is a schedule that satisfies TO rule, \(P(S)\) is acyclic (hence \(S\) is serializable).

---

**Example**

\[
\begin{array}{c|c}
\text{Transaction } T_1 & \text{Transaction } T_2 \\
\hline
\text{ts}(T_1) < \text{ts}(T_2) & \\
\hline
\hline
\text{(Node X)} & \text{(Node Y)} \\
\hline
(T_1) & (T_2) \\
\text{a} \leftarrow X & \text{d} \leftarrow Y \\
\text{X} \leftarrow \text{a}+100 & \text{Y} \leftarrow \text{2d} \\
\text{c} \leftarrow X & \text{b} \leftarrow Y \\
\text{X} \leftarrow \text{2c} & \text{Y} \leftarrow \text{b}+100 \\
\text{abort } T_1 & \text{reject!} \\
\text{abort } T_2 & \text{abort } T_1 \\
\end{array}
\]

---

**Strict T.O**

- **Problem:** Transaction reads "dirty data". Causes cascading rollbacks.
- **Solution:** Enforce "strict" schedules in addition to T.O rule

Lock written items until it is certain that the writing transaction has committed.

Use a commit bit \(C(X)\) for each element \(X\). \(C(X) = 1\) iff last transaction that last wrote \(X\) committed. If \(C(X) = 0\), delay reads of \(X\) until \(C(X)\) becomes 1.
Revisit example under strict T.O

\[ \text{ts(T}_1) < \text{ts(T}_2) \]

<table>
<thead>
<tr>
<th>(Node X)</th>
<th>(Node Y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T\textsubscript{1}) a ← X</td>
<td>(T\textsubscript{2}) d ← Y</td>
</tr>
<tr>
<td>(T\textsubscript{1}) X ← a+100</td>
<td>(T\textsubscript{2}) Y ← 2d</td>
</tr>
<tr>
<td>(T\textsubscript{2}) c ← X</td>
<td>(T\textsubscript{1}) b ← Y</td>
</tr>
</tbody>
</table>

\[ \text{ts(T}_1) < \text{ts(T}_2) \]

\[ \text{ts(T}_1) < \text{ts(T}_2) \]

\[ \text{ts(T}_1) < \text{ts(T}_2) \]

Enforcing T.O

For each element X:
- MAX\_R[X] → maximum timestamp of a transaction that read X
- MAX\_W[X] → maximum timestamp of a transaction that wrote X
- rL[X] → number of transactions currently reading X (0,1,2,...)
- wL[X] → number of transactions currently writing X (0 or 1)
- queue[X] → queue of transactions waiting on X

T.O. Scheduler

\[ r_i[X] \text{ arrives:} \]

- If \( \text{ts(T}_i) < \text{MAX}_W[X] \) abort \( T_i \)
- If \( \text{ts(T}_i) > \text{MAX}_R[X] \) then \( \text{MAX}_R[X] = \text{ts(T}_i) \)
- If (queue[X] is empty and wL[X] = 0)
  - rL[X] = rL[X]+1
  - begin \( r_i[X] \)
- Else add \( (r, T_i) \) to queue[X]

Note: If a transaction is aborted, it must be restarted with a larger timestamp. Starvation is possible.

T.O. Scheduler

\[ w_i[X] \text{ arrives:} \]

- If \( \text{ts(T}_i) < \text{MAX}_W[X] \) or \( \text{ts(T}_i) < \text{MAX}_R[X] \)
  - abort \( T_i \)
- MAX\_W[X] = \text{ts(T}_i)
- If (queue[X] is empty and wL[X]=0 AND rL[X]=0)
  - wL[X] = 1
  - begin \( w_i[X] \)
  - wait for \( T_i \) to complete
- Else add \( (w, T_i) \) to queue
**Thomas Write Rule**

MAX_R[X] MAX_W[X]

\[ \text{ts}(T_i) \]

T_i wants to write X

w[X] arrives:
- If (ts(T_i) < MAX_R[X]) abort T_i
- If (ts(T_i) < MAX_W[X]) ignore this write.
- Rest as before.....

**Optimization**

- Update MAX_R and MAX_W when operation is executed, not when enqueued. Example:
  - queue[X]:
    - W, ts=9
    - W, ts=8
    - W, ts=7
  - MAX_W[X] = 7 instead of 9
  - active write

- Multi-version timestamps
  - X:
    - Value written with ts=9
    - Value written with ts=7

**2PL ≠ T.O**

T_i: w_i[Y]
T_2: r_2[X] r_2[Y] w_2[Z] \( \text{ts}(T_2) < \text{ts}(T_3) \)
T_3: w_3[X]

Schedule S: r_2[X] w_2[X] w_1[Y] r_2[Y] w_2[Z]

**Timestamp management**

- Too much space
- Additional IOs
If a transaction reads or writes X, make entry in cache for X (add row if required).

Choose $t_{\text{MIN}} = \text{current time} - d$

Periodically purge all items X with $MAX_R[X] < t_{\text{MIN}}$ & $MAX_W[X] < t_{\text{MIN}}$ and store $t_{\text{MIN}}$.

If X has cache entry, use those $MAX_R$ and $MAX_W$ values. Otherwise assume $MAX_R[X] = MAX_W[X] = t_{\text{MIN}}$.

**Distributed T.O Scheduler**

- Each scheduler is "independent"
- At end of transaction, signal all schedulers involved, indicating commit/abort of transaction.

**Reliability**

- Correctness
  - Serializability
  - Atomicity
  - Persistence
- Availability

**Types of failures**

- Processor failures
  - Halt, delay, restart, berserk, ...
- Storage failures
  - Transient errors, spontaneous failures, persistent write errors
- Network failures
  - Lost messages, out-of-order messages, partitions

- Other ways of characterizing failures
  - Malevolent/Unintentional failures
  - Single/Multiple failures
  - Detectable/Undetectable failures
Models for Node Failure

(1) Fail-stop nodes
- Perfect
- Halted
- Recovery
- Perfect
- Volatile memory lost
- Stable storage ok

(2) Byzantine nodes
- Perfect
- Arbitrary failure
- Recovery
- Perfect
- At any given time, at most some fraction \( f \) of nodes have failed (typically \( f < 1/2 \) or \( f < 1/3 \))

Models for Network Failure

(1) Reliable network
- In order messages
- No spontaneous messages
- Timeout \( T_D \)

| If no ack in \( T_D \) sec. | Destination down |

(2) Persistent messages
- If destination is down, network will eventually deliver messages.
- Simplifies node recovery but inefficient (hides too much in network layer)

Models for Network Failure

(3) Partitionable network
- In order messages
- No spontaneous messages
- No timeouts

Scenarios

- Reliable network and Fail-stop nodes
  - No data replication (1)
  - Data replication (2)

- Partitionable network and Fail-stop nodes
  - No data replication (3)
  - Data replication (4)
**Scenario 1**

Reliable network, fail-stop nodes, no data replication

Key consequence: node N "controls" X
- N is responsible for concurrency control and recovery of X
- Single control point for each data element
- If N is down, X is unavailable

**Distributed commit problem**

Transaction T

Action: \(a_1, a_2\)
Action: \(a_3\)
Action: \(a_4, a_5\)

commit

abort

commit

**Distributed Commit**

- Make global decision on committing or aborting a distributed transaction
- Assume atomicity mechanisms at each site ensure each local component is atomic
  - Each component either commits or has no effect on local database
- Enforce rule that either all components commit or all abort

**Centralized two-phase commit**

State Transition Diagram

Coordinator

Participant

Notation: Incoming Message (* = everyone)  
Outgoing Message
Key Points

- When participant enters "W" state:
  - It must have acquired all resources (e.g. locks) required for commit
  - But, it can only commit when so instructed by the coordinator

- After sending "nok" participant can unilaterally abort.

- Coordinator enters "C" state only if all participants are in "W", i.e., it is certain that all participants will eventually commit.

Handling node failures

- Coordinator and participant logs used to reconstruct state before failure.
- Important that each message is logged before being sent
- Coordinator failure may require leader election

- Participant failure: recovery procedure depends on last log record for T
  - "C" record: commit T
  - "A" record: abort T
  - "W" record: obtain write locks for T and wait/ask coordinator or other participant
  - No log records for T: abort T

Example

<table>
<thead>
<tr>
<th>Participant log</th>
<th></th>
<th></th>
<th>crash</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1; X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>undo/redo info</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1; Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>undo/redo info</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td>&quot;W&quot; state</td>
</tr>
</tbody>
</table>

- During recovery at participant:
  - Obtain write locks for X and Y (no read locks)
  - Wait for message from coordinator (or ask coordinator)

Example: tracking who has sent "OK" msgs

<table>
<thead>
<tr>
<th>Log at coord:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>start</td>
</tr>
<tr>
<td>part=(a,b)</td>
</tr>
<tr>
<td>T1</td>
</tr>
<tr>
<td>OK</td>
</tr>
<tr>
<td>from a</td>
</tr>
<tr>
<td>RCV</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- After failure, we know still waiting for OK from node b
- Alternative: do not log receipts of "OK"s. Simply abort T1
**Presumed abort protocol**

- "F" and "A" states combined in coordinator
- Saves persistent space (forget about a transaction quicker)
- Presumed commit is analogous

**Presumed abort-coordinator** (participant unchanged)
2PC is blocking

Coordinator P1 P2 P3 P4
W W W

Case I: P1 → "W"; coordinator sent commits
P1 → "C"

Case II: P1 → NOK; P1 → A

⇒ P2, P3, P4 (surviving participants) cannot safely abort
or commit transaction

Variants of 2PC

Linear
ok ok ok
commit commit commit Coordinator

Hierarchical

Variants of 2PC

Distributed

– Nodes broadcast all messages
– Every node knows when to commit

Resources

• "Concurrency Control and Recovery" by Bernstein, Hardzilacos, and Goodman
  – Available at http://research.microsoft.com/pubs/ccontrol/

• Timestamp control
  – Chapter 9 of the CS245 Textbook ("Database System Implementation" by Garcia-Molina, Ullman, and Widom)