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


\[
\begin{array}{c|c}
\text{Node 1} & \text{Node 2} \\
\hline
X & Y \\
\end{array}
\]

\text{constraint: } X = Y

---

<table>
<thead>
<tr>
<th></th>
<th>(T_1)</th>
<th></th>
<th>(T_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(a \leftarrow X)</td>
<td>5</td>
<td>(c \leftarrow X)</td>
</tr>
<tr>
<td>2</td>
<td>(X \leftarrow a + 100)</td>
<td>6</td>
<td>(X \leftarrow 2c)</td>
</tr>
<tr>
<td>3</td>
<td>(b \leftarrow Y)</td>
<td>7</td>
<td>(d \leftarrow Y)</td>
</tr>
<tr>
<td>4</td>
<td>(Y \leftarrow b + 100)</td>
<td>8</td>
<td>(Y \leftarrow 2d)</td>
</tr>
</tbody>
</table>
**Possible Schedule**

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

If X=Y=0 initially, X=Y=200 at end

Precedence: intra-transaction (\(\downarrow\))

inter-transaction (\(\downarrow\))
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 $<_S$ where:

1. $S = \bigcup T_i$
2. $<_S \supseteq \bigcup <_i$
3. for any two conflicting operations $p, q \in S$, either $p <_S q$ or $q <_S p$

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

(T₁) \( r_1[X] \rightarrow w_1[X] \)

(T₂) \( r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X] \)

(T₃) \( r_3[X] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z] \)

\[ r_2[X] \rightarrow w_2[Y] \rightarrow w_2[X] \]

\[ S: \quad r_3[Y] \rightarrow w_3[X] \rightarrow w_3[Y] \rightarrow w_3[Z] \]

\[ r_1[X] \rightarrow w_1[X] \]
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 = \( \{ T_i \rightarrow T_j \mid \exists p \in T_i, q \in T_j \text{ such that } p, q \text{ conflict and } p <_S q \} \)

\[
\begin{align*}
S: \quad r_1[X] &\rightarrow w_1[X] &\rightarrow w_1[Y] \\
r_2[X] &\rightarrow w_2[Y] \\
r_3[X] &\rightarrow w_3[X]
\end{align*}
\]

\[
P(S): \quad T_2 &\rightarrow T_1 &\rightarrow T_3
\]
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 distributed locking](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 $X_1$ and $X_2$ on sites 1 and 2
  – T obtains read lock on $X_1$; U obtains write lock on $X_2$
  – Possible for $X_1$ and $X_2$ 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 $X$, designate specific copy $X_i$ as primary copy
- $\text{Local-lock}(X_i) \Rightarrow \text{Global-lock}(X)$

Synthesizing Global Locks

- Element $X$ with $n$ copies $X_1 \ldots X_n$
- Choose “$s$” and “$x$” such that
  - $2x > n$
  - $s + x > n$
- $\text{Shared-lock}(s \text{ copies}) \Rightarrow \text{Global-shared-lock}(X)$
- $\text{Exclusive-lock}(x \text{ copies}) \Rightarrow \text{Global-exclusive-lock}(X)$
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

\( \text{ts}(T_1) < \text{ts}(T_2) \)

(Node X)

\[(T_1) \quad a \leftarrow X\]
\[(T_1) \quad X \leftarrow a+100\]
\[(T_2) \quad c \leftarrow X\]
\[(T_2) \quad X \leftarrow 2c\]

(Node Y)

\[(T_2) \quad d \leftarrow Y\]
\[(T_2) \quad Y \leftarrow 2d\]
\[(T_1) \quad b \leftarrow Y\]
\[(T_1) \quad Y \leftarrow b+100\]

reject!

abort \(T_1\)

abort \(T_2\)
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

\(\text{ts}(T_1) < \text{ts}(T_2)\)

(Node X)

\[
\begin{align*}
(T_1) & \quad a \leftarrow X \\
(T_1) & \quad X \leftarrow a+100 \\
(T_2) & \quad c \leftarrow X \\
\end{align*}
\]

(Node Y)

\[
\begin{align*}
(T_2) & \quad d \leftarrow Y \\
(T_2) & \quad Y \leftarrow 2d \\
(T_1) & \quad b \leftarrow Y \\
\end{align*}
\]

Abort \(T_1\)

Reject!
Enforcing T.O

For each element X:

- \( \text{MAX}_R[X] \rightarrow \) maximum timestamp of a transaction that read X
- \( \text{MAX}_W[X] \rightarrow \) maximum timestamp of a transaction that wrote X
- \( \text{rL}[X] \rightarrow \) number of transactions currently reading X (0,1,2,...)
- \( \text{wL}[X] \rightarrow \) number of transactions currently writing X (0 or 1)
- \( \text{queue}[X] \rightarrow \) queue of transactions waiting on X
T.O. Scheduler

$r_i[X]$ arrives:

- If $(ts(T_i) < MAX_W[X])$ abort $T_i$
- If $(ts(T_i) > MAX_R[X])$ then $MAX_R[X] = ts(T_i)$
- 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

\(w_i[X]\) arrives:

- If \(ts(T_i) < MAX_W[X]\) or \(ts(T_i) < MAX_R[X]\)
  
  abort \(T_i\)

- \(MAX_W[X] = ts(T_i)\)

- If (queue[X] is empty and \(wL[X]=0 \text{ AND } rL[X]=0\))
  
  - \(wL[X] = 1\)
  
  - begin \(w_i[X]\)
  
  - wait for \(T_i\) to complete

- Else add \((w, T_i)\) to queue

Work out the steps to be executed when \(r_i[X]\) or \(w_i[X]\) completes.
Thomas Write Rule

MAX_R[X]  MAX_W[X]

Ts(Ti)

Ti wants to write X

w_i[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:

<table>
<thead>
<tr>
<th>queue[X]</th>
<th>W, ts=9</th>
<th>MAX_W[X] = 7 instead of 9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>W, ts=8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W, ts=7</td>
<td></td>
</tr>
</tbody>
</table>

- Multi-version timestamps

<table>
<thead>
<tr>
<th>X:</th>
<th>Value written with ts=9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value written with ts=7</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

ri[X] ts(T_i)=8
2PL ≠ T.O

Think of examples for these cases.

T₁: w₁[Y]
T₂: r₂[X] r₂[Y] w₂[Z]  ts(T₁)<ts(T₂)<ts(T₃)
T₃: w₃[X]
Schedule S: r₂[X] w₃[X] w₁[Y] r₂[Y] w₂[Z]
## Timestamp management

<table>
<thead>
<tr>
<th></th>
<th>MAX_R</th>
<th>MAX_W</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$X_n$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Too much space
- Additional IOs
Timestamp Cache

<table>
<thead>
<tr>
<th>Item</th>
<th>MAX_R</th>
<th>MAX_W</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>⋮</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- If a transaction reads or writes X, make entry in cache for X (add row if required).
- Choose $t_{s_{MIN}} \approx$ current time – d
- Periodically purge all items X with $MAX_R[X] < t_{s_{MIN}}$ & $MAX_W[X] < t_{s_{MIN}}$ and store $t_{s_{MIN}}$.
- If X has cache entry, use those $MAX_R$ and $MAX_W$ values. Otherwise assume $MAX_R[X] = MAX_W[X] = t_{s_{MIN}}$. 
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.  \rightarrow  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: \(a_1, a_2\)
Commit

Action: \(a_3\)
Abort

Action: \(a_4, a_5\)
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

\[ \text{I} \rightarrow \text{W} \rightarrow \text{A} \rightarrow \text{C} \]

- go
- exec*
- nok
- abort*
- ok*
- commit

Participant

\[ \text{I} \rightarrow \text{W} \rightarrow \text{A} \rightarrow \text{C} \]

- exec
- nok
- commit
- abort
- -

Notation: Incoming Message (\(* = \text{everyone}\))

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

### Participant log

<table>
<thead>
<tr>
<th>T1; X undo/redo info</th>
<th>⋮</th>
<th>T1; Y undo/redo info</th>
<th>⋮</th>
<th>T1 “W” state</th>
</tr>
</thead>
</table>

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

**Example:** tracking who has sent “OK” msgs

Log at coord:

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th></th>
<th>$T_1$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>start</td>
<td>part=${a,b}$</td>
<td>...</td>
<td>OK</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RCV</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<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 $T_1$
Coordinator (with timeouts and finish state)
Participant (with timeouts and finish state)

```
Participant:
  - cping, t
  - ping
  - exec
  - nok
  - commit
  - c-ok
  - abort
  - nok
  - equivalent to finish state
  - "done" message counts as either c-ok or nok for coordinator
```
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

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

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)