Last Class

- Leader election
- Distributed mutual exclusion
Decentralized Algorithm

• Use voting
• Assume n replicas and a coordinator per replica
• To acquire lock, need majority vote $m > n/2$ coordinators
  – Non blocking: coordinators returns OK or “no”
• Coordinator crash $\Rightarrow$ forgets previous votes
  – Probability that k coordinators crash $P(k) = \binom{m}{k} p^k (1-p)^{m-k}$
  – Atleast $2m-n$ need to reset to violate correctness
    • $\sum_{2m-n}^n P(k)$
Distributed Algorithm

• [Ricart and Agrawala]: needs 2(n-1) messages
• Based on event ordering and time stamps
  – Assumes total ordering of events in the system (Lamport’s clock)
• Process $k$ enters critical section as follows
  – Generate new time stamp $TS_k = TS_k + 1$
  – Send $request(k, TS_k)$ all other $n-1$ processes
  – Wait until $reply(j)$ received from all other processes
  – Enter critical section
• Upon receiving a $request$ message, process $j$
  – Sends $reply$ if no contention
  – If already in critical section, does not reply, queue request
  – If wants to enter, compare $TS_j$ with $TS_k$ and send reply if $TS_k < TS_j$, else queue
A Distributed Algorithm

a) Two processes want to enter the same critical region at the same moment.

b) Process 0 has the lowest timestamp, so it wins.

c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.
Properties

• Fully decentralized

• $N$ points of failure!

• All processes are involved in all decisions
  – Any overloaded process can become a bottleneck
A Token Ring Algorithm

(a) An unordered group of processes on a network.
(b) A logical ring constructed in software.

- Use a token to arbitrate access to critical section
- Must wait for token before entering CS
- Pass the token to neighbor once done or if not interested
- Detecting token loss in non-trivial
## Comparison

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Messages per entry/exit</th>
<th>Delay before entry (in message times)</th>
<th>Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized</td>
<td>3</td>
<td>2</td>
<td>Coordinator crash</td>
</tr>
<tr>
<td>Decentralized</td>
<td>3mk</td>
<td>2m</td>
<td>starvation</td>
</tr>
<tr>
<td>Distributed</td>
<td>(2(n-1))</td>
<td>(2(n-1))</td>
<td>Crash of any process</td>
</tr>
<tr>
<td>Token ring</td>
<td>1 to (\infty)</td>
<td>0 to (n-1)</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

- A comparison of four mutual exclusion algorithms.
Transactions

• Transactions provide higher level mechanism for *atomicity* of processing in distributed systems
  – Have their origins in databases

• Banking example: Three accounts A:$100, B:$200, C:$300
  – Client 1: transfer $4 from A to B
  – Client 2: transfer $3 from C to B

• Result can be inconsistent unless certain properties are imposed on the accesses

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A: $100</td>
<td>Read C: $300</td>
</tr>
<tr>
<td>Write A: $96</td>
<td>Write C: $297</td>
</tr>
<tr>
<td>Read B: $200</td>
<td>Read B: $200</td>
</tr>
<tr>
<td>Write B: $204</td>
<td>Write B: $203</td>
</tr>
</tbody>
</table>
ACID Properties

- **Atomic**: all or nothing
- **Consistent**: transaction takes system from one consistent state to another
- **Isolated**: Immediate effects are not visible to other (serializable)
- **Durable**: Changes are permanent once transaction completes (commits)

<table>
<thead>
<tr>
<th></th>
<th>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A:</td>
<td>$100</td>
<td></td>
</tr>
<tr>
<td>Write A:</td>
<td>$96</td>
<td></td>
</tr>
<tr>
<td>Read B:</td>
<td>$200</td>
<td></td>
</tr>
<tr>
<td>Write B:</td>
<td>$204</td>
<td></td>
</tr>
<tr>
<td>Read C:</td>
<td></td>
<td>$300</td>
</tr>
<tr>
<td>Write C:</td>
<td></td>
<td>$297</td>
</tr>
<tr>
<td>Read B:</td>
<td></td>
<td>$204</td>
</tr>
<tr>
<td>Write B:</td>
<td></td>
<td>$207</td>
</tr>
</tbody>
</table>
Transaction Primitives

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_TRANSACTION</td>
<td>Make the start of a transaction</td>
</tr>
<tr>
<td>END TRANSACTION</td>
<td>Terminate the transaction and try to commit</td>
</tr>
<tr>
<td>ABORT_TRANSACTION</td>
<td>Kill the transaction and restore the old values</td>
</tr>
<tr>
<td>READ</td>
<td>Read data from a file, a table, or otherwise</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write data to a file, a table, or otherwise</td>
</tr>
</tbody>
</table>

Example: airline reservation

Begin_transaction
  if(reserve(NY,Paris)==full) Abort_transaction
  if(reserve(Paris,Athens)==full) Abort_transaction
  if(reserve(Athens,Delhi)==full) Abort_transaction
End_transaction
Distributed Transactions

(a) A nested transaction

Subtransaction

Airline database

Hotel database

Two different (independent) databases

(b) A distributed transaction

Subtransaction

Subtransaction

Distributed database

Two physically separated parts of the same database
Implementation: Private Workspace

- Each transaction gets copies of all files, objects.
- Can optimize for reads by not making copies.
- Can optimize for writes by copying only what is required.
- Commit requires making local workspace global.

Diagram:

(a) Index

(b) Original index

(c) Private workspace

Free blocks
Option 2: Write-ahead Logs

- *In-place updates*: transaction makes changes *directly* to all files/objects
- *Write-ahead log*: prior to making change, transaction writes to log on *stable storage*
  - Transaction ID, block number, original value, new value
- Force logs on commit
- If abort, read log records and undo changes [*rollback*]
- Log can be used to rerun transaction after failure

- Both workspaces and logs work for distributed transactions
- Commit needs to be *atomic* [will return to this issue in Ch. 7]
Writeahead Log Example

\[ \begin{align*}
x &= 0; \\
y &= 0; \\
\text{BEGIN\_TRANSACTION;} \\
x &= x + 1; \\
y &= y + 2 \\
x &= y \times y; \\
\text{END\_TRANSACTION;} \\
\end{align*} \]

- a) A transaction
- b) – d) The log before each statement is executed
Concurrency Control

• Goal: Allow several transactions to be executing simultaneously such that
  – Collection of manipulated data item is left in a consistent state

• Achieve consistency by ensuring data items are accessed in an specific order
  – Final result should be same as if each transaction ran sequentially

• Concurrency control can implemented in a *layered* fashion
Concurrent Control Implementation

- General organization of managers for handling transactions.

Diagram:
- Transactions
- Transaction manager
- Scheduler
- Data manager
- BEGIN_TRANSACTION
- END_TRANSACTION
- LOCK/RELEASE
- Timestamp operations
- Execute read/write

READ/WRITE
Distributed Concurrency Control

- General organization of managers for handling distributed transactions.
Serializability

Key idea: properly schedule conflicting operations
Conflict possible if at least one operation is write
- Read-write conflict
- Write-write conflict

BEGIN_TRANSACTION
x = 0;
x = x + 1;
END_TRANSACTION

BEGIN_TRANSACTION
x = 0;
x = x + 2;
END_TRANSACTION

BEGIN_TRANSACTION
x = 0;
x = x + 3;
END_TRANSACTION

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

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

Schedule 3
x = 0;  x = 0;  x = x + 1;  x = 0;  x = x + 2;  x = x + 3;
Illegal
Optimistic Concurrency Control

- Transaction does what it wants and *validates* changes prior to commit
  - Check if files/objects have been changed by committed transactions since they were opened
  - Insight: conflicts are rare, so works well most of the time
- Works well with private workspaces
- Advantage:
  - Deadlock free
  - Maximum parallelism
- Disadvantage:
  - Rerun transaction if aborts
  - Probability of conflict rises substantially at high loads
- Not used widely
Two-phase Locking

• Widely used concurrency control technique
• Scheduler acquires all necessary locks in growing phase, releases locks in shrinking phase
  – Check if operation on data item $x$ conflicts with existing locks
    • If so, delay transaction. If not, grant a lock on $x$
    – Never release a lock until data manager finishes operation on $x$
    – One a lock is released, no further locks can be granted
• Problem: deadlock possible
  – Example: acquiring two locks in different order
• Distributed 2PL versus centralized 2PL
Two-Phase Locking

- Two-phase locking.
Strict Two-Phase Locking

- Strict two-phase locking.
Timestamp-based Concurrency Control

- Each transaction Ti is given timestamp ts(Ti)
- If Ti wants to do an operation that conflicts with Tj
  - Abort Ti if ts(Ti) < ts(Tj)
- When a transaction aborts, it must restart with a new (larger) time stamp
- Two values for each data item x
  - Max-rts(x): max time stamp of a transaction that read x
  - Max-wts(x): max time stamp of a transaction that wrote x
Reads and Writes using Timestamps

- $Read_i(x)$
  - If $ts(T_i) < max-wts(x)$ then Abort $T_i$
  - Else
    - Perform $R_i(x)$
    - $Max-rts(x) = \max(max-rts(x), ts(T_i))$

- $Write_i(x)$
  - If $ts(T_i) < max-rts(x)$ or $ts(T_i) < max-wts(x)$ then Abort $T_i$
  - Else
    - Perform $W_i(x)$
    - $Max-wts(x) = ts(T_i)$
Pessimistic Timestamp Ordering

(a) \( ts_{RD}(x) \) \( \frac{(T_1)}{\text{Time} \rightarrow} \) \( ts_{WR}(x) \) \( ts(T_2) \) \\
(b) \( ts_{WR}(x) \) \( \frac{(T_1)}{\text{Time} \rightarrow} \) \( ts_{RD}(x) \) \( ts(T_2) \) \\
(c) \( ts(T_2) \) \( \frac{(T_3)}{\text{Time} \rightarrow} \) \( ts_{RD}(x) \) \\
(d) \( ts(T_2) \) \( \frac{(T_3)}{\text{Time} \rightarrow} \) \( ts_{WR}(x) \) \\
(e) \( ts_{WR}(x) \) \( \frac{(T_1)}{\text{Time} \rightarrow} \) \( ts(T_2) \) \\
(f) \( ts_{WR}(x) \) \( \frac{(T_1)}{\text{Time} \rightarrow} \) \( ts_{tent}(x) \) \( ts(T_2) \) \\
(g) \( ts(T_2) \) \( \frac{(T_3)}{\text{Time} \rightarrow} \) \( ts_{WR}(x) \) \\
(h) \( ts(T_2) \) \( \frac{(T_3)}{\text{Time} \rightarrow} \) \( ts_{tent}(x) \)