Last Class

- Distributed Snapshots
  - Termination detection

- Election algorithms
  - Bully
  - Ring

Today: Still More Canonical Problems

- Distributed synchronization and mutual exclusion

- Distributed transactions
Distributed Synchronization

• Distributed system with multiple processes may need to share data or access shared data structures
  – Use critical sections with mutual exclusion
• Single process with multiple threads
  – Semaphores, locks, monitors
• How do you do this for multiple processes in a distributed system?
  – Processes may be running on different machines
• Solution: lock mechanism for a distributed environment
  – Can be centralized or distributed

Centralized Mutual Exclusion

• Assume processes are numbered
• One process is elected coordinator (highest ID process)
• Every process needs to check with coordinator before entering the critical section
• To obtain exclusive access: send request, await reply
• To release: send release message
• Coordinator:
  – Receive request: if available and queue empty, send grant; if not, queue request
  – Receive release: remove next request from queue and send grant
Mutual Exclusion: A Centralized Algorithm

(a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.
(b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
(c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

Properties

• Simulates centralized lock using blocking calls
• Fair: requests are granted the lock in the order they were received
• Simple: three messages per use of a critical section (request, grant, release)
• Shortcomings:
  – Single point of failure
  – How do you detect a dead coordinator?
    • A process can not distinguish between “lock in use” from a dead coordinator
      – No response from coordinator in either case
      – Performance bottleneck in large distributed systems
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 $\text{request}(k, TS_k)$ all other $n-1$ processes
  - Wait until $\text{reply}(j)$ received from all other processes
  - Enter critical section
- Upon receiving a $\text{request}$ message, process $j$
  - Sends $\text{reply}$ 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>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 ∞</td>
<td>0 to n – 1</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

• A comparison of three 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>Write B: $200</td>
</tr>
<tr>
<td>Write B: $204</td>
<td></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>Client 1</th>
<th>Client 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read A: $100</td>
<td></td>
</tr>
<tr>
<td>Write A: $96</td>
<td></td>
</tr>
<tr>
<td>Read B: $200</td>
<td></td>
</tr>
<tr>
<td>Write B: $204</td>
<td></td>
</tr>
<tr>
<td>Read C: $300</td>
<td></td>
</tr>
<tr>
<td>Write C: $297</td>
<td></td>
</tr>
<tr>
<td>Read B: $204</td>
<td></td>
</tr>
<tr>
<td>Write B: $207</td>
<td></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) Nested transaction

Subtransaction

Airline database

Hotel database

Two different (independent) databases

(b) Distributed transaction

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
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

```plaintext
x = 0;
y = 0;
BEGIN_TRANSACTION;
x = x + 1;
y = y + 2
x = x * y;
END_TRANSACTION;
```

<table>
<thead>
<tr>
<th>Log</th>
<th>Log</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = 0 / 1)</td>
<td>(x = 0 / 1)</td>
<td>(x = 0 / 1)</td>
</tr>
<tr>
<td>(y = 0/2)</td>
<td>(y = 0/2)</td>
<td>(y = 0/2)</td>
</tr>
<tr>
<td>(x = 0/4)</td>
<td></td>
<td>(x = 1/4)</td>
</tr>
</tbody>
</table>

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