Last Class

- Vector timestamps

- Global state
  - Distributed Snapshot

- Election algorithms
  - Bully algorithm

Today: Still More Canonical Problems

- Election algorithms
  - Ring algorithm

- Distributed synchronization and mutual exclusion

- Distributed transactions
Ring-based Election

- Processes have unique IDs and arranged in a logical ring
- Each process knows its neighbors
  - Select process with highest ID
- Begin election if just recovered or coordinator has failed
- Send *Election* to closest downstream node that is alive
  - Sequentially poll each successor until a live node is found
- Each process tags its ID on the message
- Initiator picks node with highest ID and sends a coordinator message
- Multiple elections can be in progress
  - Wastes network bandwidth but does no harm

A Ring Algorithm

[Diagram of a ring algorithm]
Comparison

• Assume \( n \) processes and one election in progress

• Bully algorithm
  – Worst case: initiator is node with lowest ID
    • Triggers \( n-2 \) elections at higher ranked nodes: \( O(n^3) \) msgs
  – Best case: immediate election: \( n-2 \) messages

• Ring
  – \( 2(n-1) \) messages always

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
- Process $k$ enters critical section as follows
  - Generate new time stamp $T_{Sk} = T_{Sk} + 1$
  - Send request($k$, $T_{Sk}$) 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 $T_{Sj}$ with $T_{Sk}$ and send reply if $T_{Sj} < T_{Sk}$ 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 not-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></td>
</tr>
<tr>
<td>Write A: $96</td>
<td>Read C: $300</td>
</tr>
<tr>
<td>Read B: $200</td>
<td>Write C: $297</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>Read C: $300</td>
</tr>
<tr>
<td>Read B: $200</td>
<td>Write C: $297</td>
</tr>
<tr>
<td>Write B: $204</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read B: $204</td>
</tr>
<tr>
<td></td>
<td>Write B: $207</td>
</tr>
</tbody>
</table>