Coordination 1

To do ...

- Mutual exclusion
- Election algorithms
- Next: Group communication
Process coordination

- *How can processes coordinate their action?*
- *How can they agree on a value?*

- **Two example problems**
  - Mutual exclusion – As in OS to access shared resources
  - Election of a leader or a coordinator
    (e.g., Cristian’s master goes away …)

- **Various classical algorithms for each**
  - Mutual exclusion – Central server, voting, …
  - Election – Ring-based, bully
Assumptions on failures

- Channels are reliable – “eventual” delivery
  - But unless the system is synchronous, without time bounds (hence “eventually”)

- Process fail only by crashing
  - We’ll look at arbitrary (Byzantine) failures later

- Processes are independent of each other
  - E.g., no process forwarding messages for another

*How do you know a process has failed?*
Failures and failure detectors

- To detect failures, a failure detector
  - Obviously a distributed service

- Unreliable, not always accurate
  - Given a process id, returns unsuspected/suspected
  - Just a hint, may or not be true

- Reliable
  - Unsuspected (still a hint) or failed
  - Always accurate in detecting a process has failed
  - “Crash failure” so by definition it’s not coming back!
Failures and failure detectors

- A simple solution
  - Use *heartbeats* and a *maximum transmission time*
    - You could adjust parameters at run-time
  - For synchronous system, this yields a reliable detector
  - For asynchronous – unreliable is the best you can get
    - Useful if only to solve coordination problems

- Different processes may get different responses from failure detector
  - Distributed means potentially different views
Mutual exclusion

- Processes want exclusive access to a shared resource
  - A file, a printer, …
  - Application-level protocol

\[
\begin{align*}
\text{enter()} & \quad \text{// enter critical section} \\
\text{resourceAccess()} \\
\text{exit()} & \quad \text{// exit critical section}
\end{align*}
\]

- No shared memory or single kernel support
- All communication is through message passing
Mutual exclusion

- Requirements for a solution
  - Essential requirements
    - ME1 (safety): At most one process in the CS
    - ME2 (liveness): Request to enter/exit CS are eventually granted
  - Sometimes, include a fairness requirement
    - ME3 (→ ordering): Requests to enter the CS are granted in happened-before order

- Comparing solutions
  - Bandwidth consumed – proportional to number of messages sent in entry/exit
  - Client delay – at each entry/exit operation
  - Fault tolerance
Some common algorithms

- Central server
- Ring- or token-based
- Multicast and logical clocks (Lamport’s, Ricart & Agrawala’s)
- Voting/quorum (Maekawa’s)
Mutual exclusion – Centralized

- A central server grants permission
  - Process to enter – Request to server and wait for OK
  - Server upon request – if nobody in CS, lets them go else, holds reply and queue the process
  - Server upon release – chooses *oldest* request and sends OK to process
Ring-based algorithm (token-based)

- Organize processes in a *logical ring*
- Exclusion is granted by having a token
- Token is passed around the ring
  - Don’t need it anymore? Pass it to the next

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME1 – Mutual exclusion</td>
<td>✔</td>
</tr>
<tr>
<td>ME2 – No starvation</td>
<td>✔</td>
</tr>
<tr>
<td>ME3 - Ordering</td>
<td>✗</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msgs per entry/exit</td>
<td>1 to infinite</td>
</tr>
<tr>
<td>Delay to entry (msgs)</td>
<td>0 to N-1</td>
</tr>
<tr>
<td>Problems</td>
<td>Lost token, process crash</td>
</tr>
</tbody>
</table>

And out of CS

In CS
Problem with centralized solution: ordering

Using LC – $p_i$ locally maintains $Q_i$, part of a shared priority queue

To go into critical section, $p_i$ must have replies from all others AND be at the front of $Q_i$

When it has received all replies:
- All other processes are aware of its request
- Process is aware of any earlier requests for CS
To get entry
  – Multicast request
  – Wait until getting $N-1$ replies
  – Now you have it

Upon getting a request
  – If somebody has it or another process’ request is first, queue request, else reply immediately
  – If multiple processes request entry at the same time, order by LC (id to break ties)

At exit
  – Reply to queued requests
Three processes; $p_3$ is not interested, $p_1$ and $p_2$ want to enter the critical section

Both multicast their request with timestamp 41 and 34

<table>
<thead>
<tr>
<th>ME1 – Mutual exclusion</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME2 – No starvation</td>
<td>✔</td>
</tr>
<tr>
<td>ME3 - Ordering</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>2(N-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msgs per entry/exit</td>
<td></td>
</tr>
<tr>
<td>Delay to entry (msgs)</td>
<td>2(N-1)</td>
</tr>
<tr>
<td>Problems</td>
<td>Process crash</td>
</tr>
</tbody>
</table>
Maekawa’s voting

- Don’t need everyone’s OK, just “enough” of them
- What’s enough? *Any two voting sets must have a non-empty intersection*
- Each process $p_i$ maintains a *voting set* $V_i$ ($i=1, \ldots, N$), where $V_i \subseteq \{p_1, \ldots, p_N\}$
- Sets $V_i$: chosen such that $\forall \ i, j$
  - $p_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$ (at least one common member for any two sets)
  - $|V_i| = k$ (fairness, all voting sets of the same size)
  - Each process $p_j$ is contained in $M$ of the voting sets $V_i$
Maekawa’s algorithm

- Makeawa showed that an optimal solution (i.e., a solution that minimizes $K$ while ensuring mutual exclusion)
  - $K$ (size of voting set) $\sim \sqrt{N}$ and $M = K$
- To determine $V_i$, non-trivial
  - An approximation, order processes on a grid $\sqrt{N} \times \sqrt{N}$
  - $V_i$ is row U column including $p_i$

$N = 16$
Maekawa’s voting

- For $p_i$ to enter the CS
  - Multicast request, wait for $K$ replies and you got it
- On receipt of a request from $p_i$ at $p_j$
  - If in CS or already voted, queue request w/o replying,
    else send reply and set $voted$ to true
- For $p_i$ to exit the CS
  - Set state to released and multicast this to all processes in $V_i$
- On receipt of a release from $p_i$ at $p_j$
  - If queue is non-empty, remove head of the queue and reply to originating process, set $voted$ to true
  - else set $voted$ to false
Maekawa’s voting

- As described, possible deadlock
  - \( p_1, p_2 \) and \( p_3 \) all requests entry at once
  - sets \( V_1 = \{p_1, p_2\} \), \( V_2 = \{p_2, p_3\} \), \( V_3 = \{p_3, p_1\} \)
  - \( p_1 \) replies to itself holds off \( p_2 \), \( p_2 \) replies to itself and holds off \( p_3 \), ...

- Fix: Ordering – queue outstanding requests in happened-before order

- Mutual exclusion – Since for any two processes, their voting sets intersect

<table>
<thead>
<tr>
<th>ME1 – Mutual exclusion</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>ME2 – No starvation</td>
<td>✔</td>
</tr>
<tr>
<td>ME3 - Ordering</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3√N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msgs per entry/exit</td>
<td></td>
</tr>
<tr>
<td>Delay to entry (msgs)</td>
<td>2√N</td>
</tr>
<tr>
<td>Problems</td>
<td>Crash of voting process</td>
</tr>
</tbody>
</table>
Back in 5’
Election

- Many algorithms need a process to act as coordinator
  - E.g., mutual exclusion based on central server
- In general, it doesn’t matter which one
  - So pick the one with the largest ID/weight
- Elections conclude when all agree on new coordinator
Election algorithms

- Each $p_i$ keeps id of leader in $Elected_i$ (or NIL)
- Properties to satisfy: $\forall p_i$,
  - E1 Safety: $Elected_i = $ NIL or $Elected = $ P where P is the yet non-crashed process with the largest ID
  - E2 Liveness: All $p_i$ participate and eventually either set $Elected_i \neq$ NIL or crash
- Performance measurements
  - Bandwidth – Proportional to number of msgs sent
  - Turnaround time – Number of serialized message transmissions between begin and end of a single run
A ring algorithm

- $p_i$, notice coordinator down, calls an election
- … sending an ELECTION msg, with its number in it, to first successor up
- Recipient forward msgs adding itself as candidate if its ID is $>$ than the one in the msg
- If the ID in the msg, is that of the recipient, it is the elected coordinator and inform all (ELECTED msg)
- ELECTED msgs goes around the ring once
A ring algorithm

Leader is down, I’m calling an election
A ring algorithm

- E1 is met – A process has to received its own message back before sending ELECTED around so all processes before must have lower numbers.
- E2 follows from guaranteed traversal of the rink.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Msgs</td>
<td>2(N-1)</td>
</tr>
<tr>
<td>Problems</td>
<td>Doesn’t tolerate faults</td>
</tr>
</tbody>
</table>
The Bully algorithm (Garcia-Molina)

- $p_i$ notice coordinator down, calls an election
- $p_i$ sends ELECTION msg to all processes with higher numbers
  - Assumes every process knows who those are
- If no-one responds, $p_i$ is the winner
  - Algorithm assumes a synchronous system – uses timeouts to detect process failures
- If a process with a higher number receives the ELECTION msg, reply with OK and calls an election
- When done, winner let everybody know with an ELECTED msg
The Bully algorithm

4 notice coordinator is down, calls an election, sending message to all processes with higher numbers

5 and 6 (those with higher numbers) reply with OK and calls an election

… now 5 and then 6 call an election

No-one responds, 6 is the winner and lets everyone know

If 7 ever wakes up, it will call for elections
The Bully algorithm

- E1, assuming no process is replaced, is satisfied
  - No two processes will think they are the coordinator since the one with lower number will defer to the leader
  - If crashed processes are replaced by others with same identifiers E1 is not guaranteed

- E2 works by the assumption of reliable message delivery

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Msgs</strong></td>
<td>N - 2 to O(N^2)</td>
</tr>
<tr>
<td><strong>Problems</strong></td>
<td>System must be synchronous</td>
</tr>
</tbody>
</table>
Synchronization is about doing the right thing at the right time …

What’s the right time?

What’s the right thing to do?
  – Who can access what when?
  – Who is in charge?