Today

- Mutual exclusion
- Election algorithms

Next time

- Group communication
Process coordination

- How can process coordinate their action?
- How can you make processes agree on a value?

First two example problems
- Mutual exclusion – For access share 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”)
- Processes are independent of each other
  - E.g., no process forwarding messages for another
- Process fail only by crashing
  - We’ll look at arbitrary failures later

How do you know a process has failed?
Assumptions on failures

- Failure detector – obviously a distributed service
- Unreliable – Given a process id, *Unsuspected/* suspected
  - Just a hint, may or not be true
- Reliable – *Unsuspected or failed*
  - Always accurate in detecting a process has failed
  - “Crash failure” so by definition it’s not coming back!
Assumptions on failures

- **A simple solution**
  - Use heartbeats and a maximum transmission time
    - You can also adjust parameters at run-time
  - For a synchronous system, this yields a reliable one
  - 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
  - All communication is through message passing
  - Application-level protocol

  ```
  enter() // enter critical section
  resourceAccess()
  exit() // exit critical section
  ```

- Requirements for a solution
  - Essential requirements
    - ME1 (safety): At most 1 process in the CS
    - ME2 (liveness): Request to enter/exit the CS are eventually granted
  - Sometimes, include a fairness requirement
    - ME3 (→ ordering): Requests to enter the CS are granted in happened-before order
Algorithm and metrics

- **Common algorithms**
  - Central server
  - Ring based
  - Multicast and logical clocks (Lamport’s, Ricart & Agrawala’s)
  - Voting/quorum (Maekawa’s)

- **Comparing**
  - Bandwidth consumed – proportional to number of messages sent in entry/exit
  - Client delay – at each entry/exit operation
  - Fault tolerance
Mutual exclusion – Centralized

- A central server grants permission
  - To enter – process sends request to server and waits for OK
  - Server upon request – if nobody is in CS, let them go else, hold reply and queue the process
  - Server upon release – choose *oldest* request and send OK to process

<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>

Empty queue

![Diagram showing mutual exclusion process with requests and releases from processes 0, 1, and 2]

- In CS
- Msgs per entry/exit: 2
- Delay to entry (msgs): 2
- Problems: Coordinator crash
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>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>Msgs per entry/exit</th>
<th>1 to infinite</th>
</tr>
</thead>
<tbody>
<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>
Lamport’s solution – Shared priority queue

- Problem with centralized solution: ordering

- Lamport’s solution using logical clocks – process $p_i$ locally maintains $Q_i$, part of a shared priority queue
- To go into the critical section, $p_i$ must have replies from all others AND be at the front of $Q_i$
- When it got all replies:
  - All other processes are aware of its request
  - Process is aware of any earlier requests for CS

If $p_2$ request is granted before $p_1$’s that would violate fairness (ME3)
Lamport’s solution – Shared priority queue

To enter critical section at $p_i$:
- Stamp request with the current time $T$
- Add request to $Q_i$
- Broadcast $REQUEST(T)$ to all processes
- Wait for all replies and for $T$ to reach front of $Q_i$

To leave:
- Pop head of $Q_i$, broadcast $RELEASE$ to all other

On receipt of $REQUEST(T')$ from process $p_j$:
- Add $T'$ to $Q_i$
- If waiting for $REPLY$ from $p_j$ for an earlier request $T$, wait until $p_j$ replies arrives
- Otherwise $REPLY$

On receipt of $RELEASE$
- Pop head of $Q_i$

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Msgs per entry/exit</td>
<td>3(N-1)</td>
</tr>
<tr>
<td>Delay to entry (msgs)</td>
<td>3(N-1)</td>
</tr>
<tr>
<td>Problems</td>
<td>Any process crash</td>
</tr>
</tbody>
</table>
Ricart and Agrawala

- Improvement on Lamport’s
  - Combine request/reply
  - Delay reply to any request later than your own

- 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

- At exit
  - Reply to queued requests
Ricart and Agrawala

- Three processes, \( p_1 \) and \( p_2 \) want to enter the critical section
- Both multicast their request with timestamp 41 and 34

| ME1 – Mutual exclusion | ✔ |
| ME2 – No starvation    | ✔ |
| ME3 - Ordering         | ✔ |

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

- You don’t need everyone’s OK, just “enough” of them
- What’s enough? Any two voting sets have a non-empty intersection
- Each process $p_i$ maintain a voting set $V_i$ ($i=1, ..., N$), where $V_i \subseteq \{p_1, ..., p_N\}$
- 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 of any two voting 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
  - \( K \sim \sqrt{N} \) and \( M = K \)
- To determine \( V_i \)
  - Order processes on a grid \( \sqrt{N} \times \sqrt{N} \)
  - \( V_i \) is row \( U \) column including \( p_i \)
- For \( p_i \) to enter the CS
  - Multicast request to all processes in \( V_i \)
  - Wait until you get \( K \) replies and you have it
- On receipt of a request from \( p_i \) at \( p_j \)
  - If in the CS or has already voted
    - Queue request without replying
  - Else send reply and set \( voted \) to true
Maekawa’s voting

- 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, reply to one (happened-before to avoid deadlocks) and set voted to true
  - Else set voted to false

- Mutual exclusion – Since for any two processes, their voting sets intersect
- Ordering – Used happened-before to send votes

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Msgs per entry/exit</td>
<td>$3\sqrt{N}$</td>
</tr>
<tr>
<td>Delay to entry (msgs)</td>
<td>$2\sqrt{N}$</td>
</tr>
<tr>
<td>Problems</td>
<td>Crash of voting process</td>
</tr>
</tbody>
</table>
Back in 5’
Many algorithms need a process to act as coordinator
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 process $p_i$ maintains the identity of the elected in the variable $Elected_i$ (or NIL)

- Properties to satisfy: $\forall p_i$,
  - E1 Safety: $Elected_i = \text{NIL}$ 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 $Elected_i \neq \text{NIL}$ or crash

- Performance measurements
  - Bandwidth utilization – proportional to the number of messages sent
  - Turnaround time – Number of serialized message transmissions between begin and end of a single run
A ring algorithm

- $p_i$, notice coordinator is down and calls an election
- … sending an ELECTION message, with its number in it, to first successor up
- Recipient forward messages adding itself as candidate
- Who started it all, will eventually receive a message with itself in the list; elect coordinator and inform all (ELECTED message)
- ELECTED messages goes around the ring once
A ring algorithm

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

### Election

<table>
<thead>
<tr>
<th>Msgs</th>
<th>3N-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problems</td>
<td>Doesn’t tolerate faults</td>
</tr>
</tbody>
</table>
The Bully algorithm (Garcia-Molina)

- $p_i$ notice coordinator is down and calls an election
- $p_i$ sends ELECTION message 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 message, reply with OK and calls an election
- When done, winner let everybody know with an ELECTED message
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

If 7 ever wakes up, it will call for elections

<table>
<thead>
<tr>
<th>Msgs</th>
<th>N - 2 to O(N²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problems</td>
<td>System must be synchronous</td>
</tr>
</tbody>
</table>
Summary

- Synchronization is about doing the right thing at the right time ...
- What’s the right time?
  - An issue when you don’t share clocks
- What’s the right thing to do?
  - Who can access what when?
  - Who is in charge?