Failure Assumptions and Failure Detectors

- Reliable communication channels
- Process failures: crashes
- Failure detector: object/code in a process that detects failures of other processes
- Unreliable failure detector
  - Unsuspected or suspected (evidence of possible failures)
  - Each process sends an "alive" message to everyone else
  - Not receiving an "alive" message after timeout
- Most practical systems
- Reliable failure detector
  - Unsuspected or failure
  - Synchronous system
  - Few practical systems

Distributed Mutual Exclusion

- Provide critical region in a distributed environment
- Message passing
- For example, locking files, lockd daemon in UNIX (NFS is stateless, no file-locking at the NFS level)

Algorithms for mutual exclusion

- N processes
- Processes don't fail
- Message delivery is reliable
- Critical region: enter(), resourceAccesses(), exit()
- Properties:
  - [ME1] Safety: only one process at a time
  - [ME2] Liveness: eventually enter or exit
  - [ME3] Happened-before ordering: ordering of enter() is the same as HB ordering

A central server algorithm

- Server keeps track of a token—permission to enter critical region
- A process requests the server for the token
- The server grants the token if it has the token
- A process can enter if it gets the token, otherwise waits
- When done, a process sends release and exits

Performance evaluation:
- Overhead and bandwidth consumption: # of messages sent
- Client delay incurred by a process at entry and exit
- Throughput measured by synchronization delay: delay between one's exit and next's entry
Server managing a mutual exclusion token for a set of processes

A central server algorithm
- Properties:
  - Safety, why?
  - Liveness, why?
  - HB ordering not guaranteed, why? [VC in processes vs. server]
- Performance:
  - Enter overhead: two messages (request and grant)
  - Enter delay: time between request and grant
  - Exit overhead: one message (release)
  - Exit delay: none
  - Synchronization delay: between release and grant
  - Centralized server is the bottleneck

A ring-based algorithm
- Logical ring, could be unrelated to the physical configuration
- \( p_i \) sends messages to \( p_{(i+1) \mod N} \)
- When a process holds a token, it can enter, otherwise waits
- When a process releases a token (exit), it sends to its neighbor

A ring of processes transferring a mutual exclusion token

A ring-based algorithm
- Properties:
  - Safety, why?
  - Liveness, why?
  - HB ordering not guaranteed, why?
- Performance:
  - Bandwidth consumption: token keeps circulating
  - Enter overhead: 0 to \( N \) messages
  - Enter delay: delay for 0 to \( N \) messages
  - Exit overhead: one message
  - Exit delay: none
  - Synchronization delay: delay for 1 to \( N \) messages

An algorithm using multicast and logical clocks
- Multicast a request message for the token
- Enter only if all the other processes reply
- Totally-ordered timestamps: \( <T, p_i> \)
- Each process keeps a state: RELEASED, HELD, WANTED
- If all have state = RELEASED, all reply, a process can hold the token and enter
- If a process has state = HELD, doesn’t reply until it exits
- If more than one process has state = WANTED, process with the lowest timestamp will get all \( N-1 \) replies first.
Ricart and Agrawala's algorithm

On initialization:

- state := RELEASED.
- Multicast request to all processes;
- Wait until (number of replies received = (N - 1));

To enter the section:

- state := WANTED;
- Multicast request to all processes;
- request processing deferred here
- \( T := \text{request's timestamp} \);
- Wait until (number of replies received = (N - 1));
- state := HELD;
- On receipt of a request \(<T_i, p_i>\) at \( p_j \) \((i \neq j)\):
  - if (state = HELD or (state = WANTED and \((T, p_j) < (T_i, p_i))\))
    - queue request from \( p_i \) without replying;
  - else
    - reply immediately to \( p_i \);
- end if

To exit the critical section:

- state := RELEASED;
- reply to any queued requests;

Maekawa's Voting Algorithm – Main Idea

- We actually don’t need all \( N - 1 \) replies
- Consider processes A, B, X
  - A needs replies from “A” and X
  - B needs replies from “B” and X
  - If X can only reply to (vote) *one process at a time*
    - A and B cannot have a reply from X at the same time
      - Mutex between A and B hinges on X
    - Processes in overlapping groups
      - members in the overlap “control” mutex

Mutual exclusion for all processes

- A group for every process
- A pair of groups for A and B overlaps
  - mutex(A, B)
- Every possible pair of groups overlaps
  - mutex(all possible pairs of processes)
  - mutex(all processes)

An algorithm using multicast and logical locks

- Properties
  - Safety, why?
  - Liveness, why?
  - HB ordering, why?

Multicast synchronization

- performance:
  - Bandwidth consumption: no token keeps circulating
  - Entry overhead: \( 2(N-1) \), why? [with multicast support: \( 1 + (N-1) = N \)]
  - Entry delay: delay between request and getting all replies
  - Exit overhead: 0 to \( N-1 \) messages
  - Exit delay: none
  - Synchronization delay: delay for 1 message (one last reply from the previous holder)
Groups and members

- A group for each process
  - N processes
  - N groups
- Each group has K members
  - Numbers of processes to request for permission
- Each process is in M groups (M > 1)
  - Allows overlapping => mutex
  - Number of processes to grant permission

Parameters and group membership

- N = number of processes
- K = voting group size
- M = number of voting groups each process is in
- Optimal (smallest K, why?)
  - K = M = \(\sqrt{N}\)
  - Non-trivial to construct the groups
- Approximation
  - K = \(2 \cdot \sqrt{\sqrt{N}} - 1\)
  - Put process ids in a sqrt(N) by sqrt(M) table
  - Union the rows and columns where p_i is
  - N groups
  - M = \(2 \cdot \sqrt{\sqrt{N}} - 1\)

Group membership

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>I</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
</tbody>
</table>

- Group for A: [A, B, C, D, G]
- Group for B: [A, B, C, E, H]
- ... Group for I: [G, H, I, C, F]
- Every pair of groups overlap
- N groups
- Each group has K = 2 * \(\sqrt{\sqrt{N}}\) – 1 members
- M = 2 * \(\sqrt{\sqrt{N}}\) – 1 [if of groups each process are in]

Group membership (alternative?)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>E</td>
<td>F</td>
<td>D</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>I</td>
<td>G</td>
<td>H</td>
<td>I</td>
</tr>
</tbody>
</table>

- Groups for A, B, C: [A, B, C, D, G]
- Groups for D, E, F: [D, E, F, B, H]
- Every pair of groups overlap
- N groups [\(\sqrt{\sqrt{N}}\) unique groups]
- Each group has \(K = 2 \cdot \sqrt{\sqrt{N}}\) – 1 members
- M = \(3 \cdot \sqrt{\sqrt{N}}\) for A, E, I, but M = \(6 \cdot \sqrt{\sqrt{N}}\) for the rest
- Undesirable, why?

Sketch of the Voting Algorithm

- A group of processes \(V_i\) for each process \(p_i\)
  - Each group of processes overlap
  - In groups: [A, X] and [B, X]
  - The groups for any pair of processes overlap
- To enter the critical region, \(p_i\)
  - Sends REQUEST’s to all processes in \(V_i\)
  - Waits for REPLY’s (VOTE)’s from all processes in \(V_i\)
- To exit the critical region, \(p_i\)
  - Sends RELEASE’s to all processes in \(V_i\)
  - Three types of messages, not two as in the multicast alg.

Figure 12.6

Maekawa’s voting algorithm

On initialization
state = RELEASED;
 voted = FALSE;
For \(p_i\) to enter the critical section
state := WANTED;
Multicast request to all processes in \(V_i\);
Wait until (number of replies received = K);
state := HELD;
On receipt of a request from \(p, p_i\) at \(p_j\)
if (state = HELD or voted = TRUE)
then
queue request from \(p_i\) without replying;
else
send reply to \(p_j\);
end if
voted := TRUE;
end if
For \(p_i\) to exit the critical section
state := RELEASED;
Multicast release to all processes in \(V_i\);
On receipt of a release from \(p, p_i\)
if (queue of requests is non-empty)
remove head of queue from \(p_j\), say;
send reply to \(p_j\);
voted := FALSE;
else
end if
Deadlock?

- Processes: A, B, C
  - Group A: A, B
  - Group B: B, C
  - Group C: C, A

- Deadlock
  - A has A’s reply, waiting for B’s reply
  - B has B’s reply, waiting for C’s reply
  - C has C’s reply, waiting for A’s reply

- Timestamp the requests in HB ordering
  - Holding according to the timestamp

Properties

- Safety: No process can reply/vote more than once at any time.
- Liveness: timestamp (HB ordering)
- HB ordering: above

Performance

- Entry overhead [assuming \( k = \sqrt{N} \)]
  - \( \sqrt{N} \) requests + \( \sqrt{N} \) replies
  - \( 2^* \sqrt{N} \)
  - \( \leq 2(N - 1) \) [\( N > 4 \)]

- Exit overhead
  - \( \sqrt{N} \) releases

Elections

- Choosing a unique process for a particular role
- For example, server in dist. mutex
- Each process can call only one election
- Multiple concurrent elections can be called by different processes
- Participant: engages in an election
- Process with the largest id wins
- Each process \( p_i \) has variable \( elected_i = ? \) (don’t know) initially

- Properties:
  - [E1] \( elected \) of a "participant" process must be \( P_{\text{max}} \)
    (elected process—largest id) or \( ? \)
  - [E2] Liveness: all processes participate and eventually set \( elected_i = ? \) (or crash)

- Performance:
  - Overhead (bandwidth consumption): # of messages
  - Turnaround time: # of messages to complete an election

A ring-based algorithm

- Logical ring, could be unrelated to the physical configuration
- \( p_i \) sends messages to \( P_{(i+1) \mod N} \)
- No failures
- Elect the coordinator with the largest id
- Initially, every process is a non-participant
- Any process can call an election:
  - Marks itself as participant
  - Places its id in an election message
  - Sends the message to its neighbor
Ring-based algorithm

- receiving an **election** message:
  - if id > myid, forward the msg, mark participant
  - if id < myid
    - non-participant: replace id with myid, forward the msg, mark participant
    - participant: stop forwarding (why? Later, multiple elections)
  - if id = myid, coordinator found, mark non-participant, elected, := id, send elected message with myid

- receiving an **elected** message:
  - if id = myid, mark non-participant, elected, := id forward the msg
  - if id = myid, stop forwarding

A ring-based election in progress

Note: The election was started by process 17. The highest process identifier encountered so far is 24. Participant processes are shown darkened.

Ring-based algorithm

- Properties:
  - safety: only the process with the largest id can send an elected message
  - liveness: every process in the ring eventually participates in the election; extra elections are stopped

Performance:

- one election, best case, when?
  - 2N election messages
  - 2N elected messages
  - turnaround: 2N messages

- one election, worst case, when?
  - 2N - 1 election messages
  - 2N - 1 elected messages
  - turnaround: 3N - 1 messages

- can’t tolerate failures, not very practical

The bully algorithm

- processes can crash and can be detected by other processes
- timeout \( T = 2T_{\text{trans}} + T_{\text{proc}} \)
- each process knows all the other processes and can communicate with them
- Messages: election, answer, coordinator

The bully algorithm

- start an election
  - detects the coordinator has failed
  - sends an election message to all processes with higher id’s and waits for answers (except the failed coordinator/process)
  - if no answers in time \( T \),
    - it is the coordinator
    - sends coordinator message (with its id) to all processes with lower id’s
  - else
    - waits for a coordinator message
    - starts an election if timeout
The bully algorithm

The election of coordinator \( p_2 \), after the failure of \( p_3 \) and then \( p_4 \)

Stage 1

Stage 2

Stage 3

Stage 4

Eventually...

The election of coordinator \( p_2 \) after the failure of \( p_4 \) and then \( p_3 \).

The bully algorithm

Receiving an election message
- Sends an answer message back
- Sends an election message to all higher-id processes (including the “failed” coordinator—the coordinator might be up by now)

Receiving a coordinator message
- Set elected to the new coordinator
- To be a coordinator, it has to start an election
- When a crashed process is replaced
  - The new process starts an election and
  - Can replace the current coordinator (hence “bully”)

The bully algorithm

Properties:
- Safety:
  - A lower id process always yields to a higher-id process
  - However, during an election, if a failed process is replaced
    - the low-id processes might have two different coordinators: the newly elected coordinator and the new process, why?
  - Failure detection might be unreliable
- Liveness: all processes participate and know the coordinator at the end

The bully algorithm

Performance
- Best case: when?
  - Overhead: \( N/2 \) coordinator messages
  - Turnaround delay: no election/answer messages
- Worst case: when?
  - Overhead:
    - \( 1 + 2 + \ldots + (N-2) + (N-2) = (N-1)(N-2)/2 + (N-2) \) election messages,
    - \( 1 + \ldots + (N-2) \) answer messages,
    - \( N/2 \) coordinator messages,
  - Total: \( (N-1)(N-2)/2 + 2(N-2) = (N+1)(N-2)/2 = O(N^2) \)
- Turnaround delay: delay of election and answer messages