Distributed Systems

Mutual Exclusion & Election Algorithms

Paul Krzyzanowski
pxk@cs.rutgers.edu

Except as otherwise noted, the content of this presentation is licensed under the Creative Commons Attribution 2.5 License.
Mutual Exclusion & Election Algorithms
Process Synchronization

Techniques to coordinate execution among processes

- One process may have to wait for another
- Shared resource (e.g. critical section) may require exclusive access
Centralized Systems

Mutual exclusion via:
- Test & set in hardware
- Semaphores
- Messages
- Condition variables
Distributed Mutual Exclusion

- Assume there is agreement on how a resource is identified
  - Pass identifier with requests
- Create an algorithm to allow a process to obtain exclusive access to a resource.
Centralized algorithm

- Mimic single processor system
- One process elected as coordinator

1. Request resource
2. Wait for response
3. Receive grant
4. *access resource*
5. Release resource

![Diagram showing a centralized algorithm with a process (P) communicating with a coordinator (C) through request and grant actions.](image-url)
Centralized algorithm

If another process claimed resource:
- Coordinator does not reply until release
- Maintain queue
  - Service requests in FIFO order
Centralized algorithm

Benefits

• Fair
  – All requests processed in order
• Easy to implement, understand, verify

Problems

• Process cannot distinguish being blocked from a dead coordinator
• Centralized server can be a bottleneck
Token Ring algorithm

Assume known group of processes
- Some ordering can be imposed on group
- Construct logical ring in software
- Process communicates with neighbor
Token Ring algorithm

• Initialization
  - Process 0 gets token for resource R

• Token circulates around ring
  - From \( P_i \) to \( P_{(i+1) \mod N} \)

• When process acquires token
  - Checks to see if it needs to enter critical section
  - If no, send ring to neighbor
  - If yes, access resource
    • Hold token until done

\( \text{token}(R) \)
Token Ring algorithm

• Only one process at a time has token
  - Mutual exclusion guaranteed

• Order well-defined
  - Starvation cannot occur

• If token is lost (e.g. process died)
  - It will have to be regenerated

• Does not guarantee FIFO order
  - sometimes this is undesirable
Ricart & Agrawala algorithm

- Distributed algorithm using reliable multicast and logical clocks

- Process wants to enter critical section:
  - Compose message containing:
    - Identifier (machine ID, process ID)
    - Name of resource
    - Timestamp (totally-ordered Lamport)
  - Send request to all processes in group
  - Wait until everyone gives permission
  - Enter critical section / use resource
Ricart & Agrawala algorithm

• When process receives request:
  - If receiver not interested:
    • Send OK to sender
  - If receiver is in critical section
    • Do not reply; add request to queue
  - If receiver just sent a request as well:
    • Compare timestamps: received & sent messages
    • Earliest wins
    • If receiver is loser, send OK
    • If receiver is winner, do not reply, queue

• When done with critical section
  - Send OK to all queued requests
Ricart & Agrawala algorithm

- N points of failure
- A lot of messaging traffic
- Demonstrates that a fully distributed algorithm is possible
Lamport’s Mutual Exclusion

Each process maintains request queue
- Contains mutual exclusion requests

Requesting critical section:
- Process $P_i$ sends $\text{request}(i, T_i)$ to all nodes
- Places request on its own queue
- When a process $P_j$ receives a request, it returns a timestamped $\text{ack}$
Lamport’s Mutual Exclusion

**Entering critical section** *(accessing resource)*:

- $P_i$ received a message *(ack or release)* from every other process with a timestamp larger than $T_i$
- $P_i$’s request has the earliest timestamp in its queue

**Difference from Ricart-Agrawala:**

- Everyone responds *(acks)* … always - no hold-back
- Process decides to go based on whether its request is the earliest in its queue
Lamport’s Mutual Exclusion

**Releasing critical section:**

- Remove request from its own queue
- Send a timestamped *release* message

- When a process receives a *release* message
  - Removes request for that process from its queue
  - This may cause its own entry have the earliest timestamp in the queue, enabling it to access the critical section
Election algorithms
Elections

• Need one process to act as coordinator

• Processes have no distinguishing characteristics

• Each process can obtain a unique ID
Bully algorithm

- Select process with largest ID as coordinator
- When process P detects dead coordinator:
  - Send *election* message to all processes with higher IDs.
    - If nobody responds, P wins and takes over.
    - If any process responds, P’s job is done.
  - Optional: Let all nodes with lower IDs know an election is taking place.
- If process receives an *election* message
  - Send *OK* message back
  - Hold election (unless it is already holding one)
Bully algorithm

- A process announces victory by sending all processes a message telling them that it is the new coordinator.

- If a dead process recovers, it holds an election to find the coordinator.
Ring algorithm

- Ring arrangement of processes
- If any process detects failure of coordinator
  - Construct *election* message with process ID and send to next process
  - If successor is down, skip over
  - Repeat until a running process is located
- Upon receiving an election message
  - Process forwards the message, adding its process ID to the body
Ring algorithm

Eventually message returns to originator
- Process sees its ID on list
- Circulates (or multicasts) a coordinator message announcing coordinator
  - E.g. lowest numbered process
Problems with elections

Network segmentation
  - Split brain

Rely on alternate communication mechanism
  - Redundant network, shared disk, serial line, SCSI
The end.