Distributed Systems
09. Consensus: Mutual Exclusion & Election Algorithms

Paul Krzyzanowski
Rutgers University
Spring 2020

March 25, 2020
© 2014-2020 Paul Krzyzanowski

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

Mutual exclusion
- Examples
  - Update fields in database tables
  - Modify a file
  - Modify file contents that are replicated on multiple servers
- Easy to handle if the entire request is atomic
  - Contained in a single message; server can manage mutual exclusion
- Needs to be coordinated if the request comprises multiple messages or spans multiple systems

March 25, 2020
© 2014-2020 Paul Krzyzanowski

Centralized Systems
Achieve mutual exclusion via:
- Test & set in hardware
- Semaphores
- Messages (inter-process)
- Condition variables

March 25, 2020
© 2014-2020 Paul Krzyzanowski

Distributed Mutual Exclusion
Goal:
Create an algorithm to allow a process to request and obtain exclusive access to a resource that is available on the network.

Required properties:
Safety: At any instant, only one process may hold the resource
Liveness: The algorithm should make progress; processes should not wait forever for messages that will never arrive
Fairness: Each process gets a fair chance to hold the resource: bounded wait time & in-order processing

March 25, 2020
© 2014-2020 Paul Krzyzanowski

Assumption
Assume there is agreement on how a resource is identified
- Pass the identifier with requests
  - e.g., lock("printer")
  - lock("table:employees"),
  - lock("table:employees;row:15")

…and every process can identify itself uniquely

We’ll just use request(R) to request exclusive access to resource R.

March 25, 2020
© 2014-2020 Paul Krzyzanowski

Categories of algorithms
- Centralized
  - A process can access a resource because a central coordinator allowed it to do so

- Token-based
  - A process can access a resource if it is holding a token permitting it to do so

- Contention-based
  - A process can access a resource via distributed agreement

March 25, 2020
© 2014-2020 Paul Krzyzanowski
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

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

Benefits
- Fair: All requests processed in order
- Easy to implement, understand, verify
- Processes do not need to know group members – just the coordinator

Problems
- Process cannot distinguish being blocked from a dead coordinator – single point of failure
- Centralized server can be a bottleneck

Token Ring algorithm

Assume known group of processes
- Some ordering can be imposed on group (unique process IDs)
- Construct logical ring in software
- Process communicates with its neighbor

Initialization
- Process 0 creates a 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

Your turn to access resource R
Token Ring algorithm

$P_0$ $P_1$ $P_2$ $P_3$ $P_4$ $P_5$

Your turn to access resource $R$
**Token Ring algorithm summary**

- Only one process at a time has token
  - Mutual exclusion guaranteed
- Order well-defined (but not necessarily first-come, first-served)
  - Starvation cannot occur
  - Lack of FCFS ordering may be undesirable sometimes
- Problems
  - Token loss (e.g., process died)
    - It will have to be regenerated
  - Detecting loss may be a problem (is the token lost or in just use by someone?)
  - Process loss: what if you can’t talk to your neighbor?

**Lamport’s Mutual Exclusion**

Distributed algorithm using reliable multicast and logical clocks

- Each process maintains request queue
  - Queue contains mutual exclusion requests
- Messages are sent reliably and in FIFO order
  - Each message is time stamped with totally ordered Lamport timestamps
  - Ensures that each timestamp is unique
  - Every node can make the same decision by comparing timestamps
- Queues are sorted by message timestamps

**1. Request a Resource**

**Request a critical section:**

- Process $P_i$ sends request($i$, $T_i$) to all nodes
  - It also places the same request onto its own queue
- When a process $P_j$ receives a request:
  - It returns a timestamped ack
  - Places the request on its request queue
- Every process will have an identical queue
  - Same contents in the same order

**2. Use the Resource**

**Enter a critical section (accessing resource):**

- $P_i$ has received ACKs from everyone
- $P_i$’s request has the earliest timestamp in its queue

If your request is at the head of the queue AND you received ACKs for that request ... you can access the critical section

**3. Release the resource**

**Release a critical section:**

- Process $P_i$ removes its request from its queue
- Sends release($i$, $T_j$) to all nodes
- Each process now checks if its request is the earliest in its queue
- If so, that process now has the critical section
Lamport’s Mutual Exclusion

- **Performance**
  - $3(N - 1)$ messages per critical section
  - $(N - 1)$ Request msgs + $(N - 1)$ Reply msgs + $(N - 1)$ Release msgs
- **N points of failure**
- **A lot of messaging traffic**
  - Requests & releases are sent to the entire group
- **Not great … but demonstrates that a fully distributed algorithm is possible**

Ricart & Agrawala algorithm

Another distributed algorithm using reliable multicast and logical clocks

When a process wants to enter critical section:
1. **Compose message** containing:
   - Identifier (machine ID, process ID)
   - Name of resource
   - Timestamp (e.g., totally-ordered Lamport)
2. **Reliably multicast** request to all processes in group
3. **Wait** until everyone gives permission
4. **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: (potential race condition)
  - Compare timestamps on received & sent messages
  - Earliest wins
  - If receiver is the loser: send OK
  - If receiver is the winner: do not reply – queue the request
- **When done** with critical section
  - Send OK to all queued requests

Lamport vs. Ricart & Agrawala

**Lamport**
- Everyone responds (acks) … always – no hold-back
- $3(N - 1)$ messages
  - Request – ACK – Release
- Process decides to go based on whether its request is the earliest in its queue

**Ricart & Agrawala**
- If you are in the critical section (or won a tie)
  - Don’t respond with an ACK until you are done with the critical section
- $2(N - 1)$ messages
  - Request – ACK
- Process decides to go if it gets ACKs from everyone

Election algorithms
Elections

• Purpose
  – Need to pick one process to act as coordinator

• Processes have no distinguishing characteristics

• Each process has a unique ID to identify itself

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:
  – Sends 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

Rule: highest ID process is the leader
Suppose P_i dies
P_i detects P_j is not responding

Example: P_2 detects that P_5 is not responding
P_2 starts an election
Contacts all higher-numbered systems

Everyone who receives an ELECTION message responds
... and holds their own election, contacting higher ID processes

Example: P_3 receives the message from P_2
Responds to P_3
Sends ELECTION messages to P_4 and P_5
**Bully algorithm**

$P_i$ responds to $P_0$ and $P_i$'s messages … and holds an election

Nobody responds to $P_i$.

After a timeout, $P_i$ declares itself the leader

---

**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

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

---

Assume $P_i$ discovers that the coordinator, $P_5$, is dead
$P_i$ starts an election

Election: $\{P_1, P_2\}$
Ring algorithm

P5 RECEIVES THE ELECTION MESSAGE THAT IT INITIATED
P5 NOW PICKS A LEADER (E.G., LOWEST OR HIGHEST ID)

P0 receives the election message that it initiated
P0 now picks a leader (e.g., lowest or highest ID)

P3 receives the election message that it initiated
P3 now picks a leader (e.g., lowest or highest ID)

P2 receives the election message that it initiated
P2 now picks a leader (e.g., lowest or highest ID)

P1 receives the election message that it initiated
P1 now picks a leader (e.g., lowest or highest ID)

P4 announces that P4 the new coordinator to the group
Chang & Roberts Ring Algorithm

**Optimize the ring**
- Message always contains one process ID
- Avoid multiple circulating elections
- If a process sends a message, it marks its state as a participant

- Assume highest # PID is the winner

**Chang & Roberts Ring Algorithm**

Upon receiving an election message:
- If PID(message) > PID(process)
  - forward the message
- If PID(message) < PID(process)
  - replace PID in message with PID(process)
  - forward the new message
- If PID(message) < PID(process) AND process is participant
  - discard the message
- If PID(message) == PID(process)
  - the process is now the leader
  - message fully circulated to the one who started: announce winner

Network Partitions: Split Brain

- Network partitioning (segmentation)
- Split Brain
- Multiple nodes may decide they're the leader

- Dealing with partitioning
  - Insist on a majority → if no majority, the system will not function
  - Rely on alternate communication mechanism to validate failure
  - Redundant network, shared disk, serial line, SCSI
- We will visit this problem later!

The End