Mutual Exclusion

Leader Election

Why Leader Election

- Given a group of processes, we want to elect a leader that is a "special" designated process for certain tasks
  - Who is the primary replica?
  - Useful for implementing centralized algorithms, since leader can broadcast messages to keep replicas in sync

- All processes must agree on who the leader is
- Any process can call for an election at any time
- A process can call for only one election at a time
- Multiple processes can call for an election simultaneously
- Result of the election should not depend on which process calls for it

Chang-Roberts Leader Election

- Processes arranged in a ring, first phase:

1. To start an election, send your id clockwise as part of "election" message
2. If received id is greater than your own, send the id clockwise
3. If received id is smaller, send your id clockwise
4. If received id is equal, then you are the leader (we assume unique id's)

Second phase:

1. Leader sends an "elected" message along with id
2. Other processes forward it and can leave the election phase

Analysis

- Worst-case: 3N-1 messages
- N-1 messages for everyone to circulate their value
- N messages for election candidate to be confirmed
- N ‘elected’ messages to announce the winner

Locks

- Only one process allowed to execute the critical section at any given time
- Non-distributed settings: solved using locks or semaphores
  - Both these approaches used shared variables
  - Not directly applicable in distributed settings where message-passing is the sole communication mechanism

Requirements

- Safety: At any instant, only one process can execute the critical section
  - Nothing bad ever happens
- Liveness: Absence of deadlock and starvation. Processes should not wait endlessly to enter the critical section
  - Something good eventually happens
- Fairness: Processes get a fair chance to enter the CS.
  - Usually, CS requests are granted on the order of their arrival

Metrics

- Message complexity: #messages exchanged per CS execution
- Synchronization delay: Time required before the next process enters the CS
- Response time: Time required between intial request and entering the CS
- Throughput: 1/(sync-delay+critical-sec-time)

Token Based

- Similar to leader election
- Processes arranged in a ring and pass a "token"
- If token rcvd && dont want to enter CS → Pass token

Centralized

- Assume leader exists
- To enter CS, seek permission from leader

Lamport's Algorithm

- Similar to totally ordered multicast
- Requests to enter the CS are timestamped and broadcast
- Processes maintain a request queue

Lamport's Mutual Exclusion Algorithm

- Requesting the CS:
    1. If P_i wants to enter the CS, it broadcasts a Request message (ts,i) and places the request on its own request queue
    2. All processes place the request in their queue, ordered by timestamp, and send an ack to P_i
- Executing the CS: Process-i enters the CS when the following two conditions hold:
    1. P-i has received a message with timestamp larger than ts from all processes
    2. P-i's request is at the head of the request queue
- Releasing the CS:
    1. Remove request from queue and broadcast a timestamped Release message
    2. When process-j receives a release message, it deletes P-i's request from its queue

Correctness proof

- Proof by contradiction
- Suppose P_i and P_j enter the CS at the same time.
- This implies that at some point in time (t), both P_i and P_j had their own requests at the top of their respective queues
- Assume the timestamp of P_i is smaller than P_j . Recall that lamport timestamps can be totally ordered .
- This means that when P-i's request message was present in P-j's request queue, and P-j was already in the CS.
- But request queues are ordered by timestamps, and P-I's is smaller
- Assumes FIFO ordering of messages between proceses

Performance

- For each CS execution, need N-1 request messages, N-1 replies, and N-1 release

Quorum based

- Processes do not request permission from all other sites, but only a subset
- Every pair of processes has a processes that mediates conflicts between that pair
- Processes can send only one reply message at any time, and only after it has received a release message for the previous reply message
- Quorums must be mutually pairwise intersecting
- Quorums cannot contain complete subsets