L05b. Lamport Clocks

Introduction:

- Each node in a distributed system knows:
  - Its own events.
  - Its communication events.
- Lamport's logical clocks:
  - The idea here is that we need to associate a time stamp with each event in the entire distributed system.
  - We'll have a local clock (counter) attached to each process. The time stamp would be the counter value.
  - The counter value is monotonically increasing.
  - The time stamp of communication events will be either the counter value of the sender process or the receiver process whichever greater.

Logical Clock Conditions:

- If we have two events a and b in the same process i, then $C_i(a) < C_i(b)$.
- If we have a communication event between event a on process i and event d on process j then:
  - $C_i(a) < C_j(d)$
  - $C_j(a) = \max(C_i(a)++, C_j)$
- If we have two concurrent events b and d, then the time stamps will be arbitrary.
- This means that Lamport Clocks gives us a partial order of all the events happening on the distributed system.

Lamport Total Order:

- If we have two events a process i and b on process j, and we can to assert that a is totally ordered ahead of b
  ($a \Rightarrow b$) if and only if:
  - $C_i(a) < C_j(d)$ or
  - $C_i(a) = C_j(d)$ and $P_i \ll P_j$, where ($\ll$) is and arbitrary well-known condition to break the tie (e.g. the greater the process ID the higher the order).
- Once we get the total order, time stamps are meaningless.

Distributed Mutual Exclusion (ME) Lock Algorithm:

- In a distributed system, we don't have a shared memory to facilitate lock implementation. So we'll use Lamport Clocks to implement a lock.
- Every process will have a queue data structure ordered by the "Happened Before" relationship.
- Any process that needs to acquire the lock will send a message to all the other processes with its time stamp as the request time.
- Each other process will put the request in its queue, and then acknowledges the request.
- If we have a tie (Two processes sent the same time stamp), we break it by giving priority to the process with higher ID.
- A process knows that it has the lock if:
  - Its request is on top of the queue.
  - It has already received acknowledges from the other processes.
- Whenever a process wants to release a lock, it sends an unlock message to all other processes.
- When the other processes receive the unlock message, they remove the sending process entry from their queues.
- The algorithm assumptions:
  - Message arrive in order.
  - There's no message loss.
  - The queues are totally ordered.
- Message complexity:
  - $N - 1$ request messages.
  - $N - 1$ acknowledge messages.
  - $N - 1$ unlock messages.
  - $Total = 3(N - 1)$.
  - Can we do better?
    If a process i lock request precedes another process j request in the queue, we can defer the acknowledgement of i and use the unlock message itself as an acknowledgement for j.

Lamport Physical Clock:

- In real world scenarios, the logical clock might be drifting of the real time due to anomalies in the logical clocks.
- We can say the event a happened before event b in real time ($a \longmapsto b$) if:
  - $C_i(a) < C_i(b)$
  - Physical conditions:
    1. PC1: Bound on individual clock drift:
    $$\left(\frac{dC_i(t)}{dt} - 1\right) < k \quad \forall i; \ (k \ll 1)$$
    2. PC2: Bound on mutual drift:
    $$\forall i, j: C_i(t) - C_j(t) \ll \epsilon$$

- IPC time and clock drift:
  - If μ is the lower bound on IPC, to avoid anomalies when:

    $$a_i \mapsto b_i$$

    1. $C_i(t+\mu) - C_j(t) > 0$
    2. $C_i(t+\mu) - C_i(t) > \mu(1-k)$

    Using equations 1 and 2, and bound ϵ on mutual drift:

    $$\mu \geq {}^{\epsilon}/_{(1-k)}$$