COMS 552 HW4

Problem 1 Lamport's DME algorithm [25pts]

Develop an example to show that, in Lamport's DME algorithm, if a site Si is executing the critical section, then Si's request need not be at the top of the request queue at another site Sj.

- Setup
  - Consider an example with three sites: S1, S2, S3 with respective queues Q1, Q2, Q3
- Execution
  - S1 requests and enters CS
    - S1 broadcasts the request with timestamp (1, S1)
    - S2 and S3 receive it. They put it in their queues and reply
    - S1 enter the CS
    - State of queues
      - Q1: [(1,S1)]
      - Q2: [(1,S1)]
      - Q3: [(1,S1)]
  - S2 requests CS while S1 is still executing
    - S2 broadcasts the request with timestamp (5, S2)
    - S2 and S3 receive it. They put it in their queues and reply. The reply has a timestamp > 5
    - State of queues
      - Q1: [(1,S1), (5,S2)]
      - Q2: [(1,S1), (5,S2)]
      - Q3: [(1,S1), (5,S2)]
  - S1 exits and releases CS
    - S1 exits the CS and removes its own request (1, S1) from Q1
    - S1 sends a timestamped release message to S2 and S3
    - State of queues
      - Q1: [(5,S2)]
      - Q2: [(1,S1), (5,S2)]
      - Q3: [(1,S1), (5,S2)]
  - Processing RELEASE
    - At S2, the RELEASE message from S1 arrives immediately
      - S2 removes (1, S1) from Q2
      - S2 checks for CS conditions
        - S2 has received messages with timestamp > 5 from S1 and S3, so this holds
        - (5, S2) is at the top of Q2 so this holds
      - So S2 can enter the CS
    - At S3, the RELEASE message from S1 is delayed
      - S3 has not processes S1's release
      - So Q3 is still [(1,S1), (5,S2)]
    - State of queues
      - Q1: [(5,S2)]
      - Q2: [(5,S2)]

        - Q3: [(1,S1), (5,S2)]
    - Result
        - So we see that S2 is in the critical section
        - At S3, Q3 is [(1,S1), (5,S2)]
        - We found an example where Si is in the CS, but Si's request is not at the top of the request queue at another site Sj

Problem 2 Ricart-Agrawala's DME algorithm [25pts]

Prove that in the Ricart-Agrawala algorithm, the critical section is accessed according to the increasing order of timestamps.

- We do a proof by contradiction
- **Proof By Contradiction**
    - Assumption
        - We assume that there are two sites Si and Sj that both request access to the CS
        - Let the timestamp of Si's request be smaller than timestamp of Sj's request
        - Assume for contradiction that Sj enters the CS before Si
    - Based on the algorithm, Si needs to receive a REPLY from all other sites (including itself), to enter the CS
    - When Si receives the REQUEST from Sj, Si decides whether to send a REPLY immediately or defer it
    - Si only send a REPLY if:
        - Si is neither requesting or in the CS or
        - Si is requesting, but Sj's timestamp is smaller than Si's
    - We see that a contradiction is formed
        - We know that Si is requesting the CS from our assumption
        - We also know timestamp of Si < timestamp of Sj
        - Therefore the condition for the immediate reply isn't satsified
        - So Si defers the reply to Sj
    - Since Si defers the reply, Sj cannot receive all of the necessary REPLY messages and therefore cannot enter the CS before Si
    - So we have shown that it impossible for Sj to enter the CS before Si when Si's timestamp < Sj's timestamp
    - Therefore the critical section must accessed according to the increasing timestamp order

Problem 3 Maekawa's DME algorithm [25pts]

Recall Maekawa's algorithm for distributed mutual exclusion and answer the following question. Consider a system with N=3 sites and K=2. The request sets of these sites are R1={1,2}, R2={2,3} and R3={3,1}. Suppose each site needs to enter its critical section for one time. Describe a sequence in which these requests are sent and processed such that totally 23 messages have to be sent. Here, each message that is sent to the sender itself is counted; for example, according to the protocol, requesting site 1 should send request messages to itself and site 2 as well as receive 2 reply message from itself and site 2 eventually, which count 4.

- We will describe a sequence where 3 sites (N = 3, K = 2) enter the CS exactly once with a total of 23 messages
- To generate the extra messages needed to reach 23, we will introduce a deadlock

- Sequence
  - Forming a deadlock
    - Sending requests
      - S1 sends REQUEST(1) to S1 and S2
      - S2 sends REQUEST(2) to S2 and S3
      - S3 sends REQUEST(3) to S3 and S1
      - Total messages so far: 6
    - Initial locks
      - S1 receives S1's request first. Sends REPLY(1) to S1, locking S1
      - S2 receives S2's request first. Sends REPLY(2) to S2, locking S2
      - S3 receives S3's request first. Sends REPLY(3) to S3, locking S3
      - Total messages so far: 9
  - Resolving the deadlock
    - Now the send requests arrive at sites that are already locked
    - Let us also assume that timestamp S1 < timestamp S2 < timestamp S3, defining the priority of the messages
    - S1 conflict
      - S3 (lowest priority) arrives at S1 (locked by highest priority)
      - So S1 sends FAILED(1) to S3
    - S2 conflict
      - S2 (medium priority) arrives at S3 (locked by lowest priority)
      - Since S2 has higher priority, S3 send INQUIRE(3) to S3
    - S3 conflict
      - S1 (highest priority) arrives at S2 (locked by medium priority)
      - Since S1 has higher priority, S2 send INQUIRE(2) to S2
    - S3 yields
      - S3 received an INQUIRE and a FAILED
      - So it meets the condition to yield, sending YIELD(3) to S3
    - Total messages so far: 13
  - S2 enters and exits CS
    - Now that S3 is yielding, S3 can process the waiting request from S2
    - After processing S2's request, it sends REPLY(3)
    - S2 now holds locks on S2 and S3, and enters CS
    - S2 exits the CS and sends RELEASE(2) to S2 and S3
    - When S3 processes S2's RELEASE, it finds S3's request in its queue, sending REPLY(3) to S3
    - Total messages so far: 17
  - S1 enters and exits CS
    - When S2 releases S2, it becomes available to S1 who was waiting
    - S2 sends REPLY(2) after processing S1's request
    - S1 now holds locks on S1 and S2, and enter CS
    - S1 exists CS and sends RELEASE(1) to S1 and S2
    - Total messages so far: 20
  - S3 enters and exits CS
    - Now there are resources available for S3
    - S1 sends REPLY(1) to S3
    - S3 now holds lock on S3 and S1, and enters CS

        - S3 exits CS and sends RELEASE(3) to S3 and S1
        - Total messages so far: 23
- So we see that we have described a sequence with 23 messages as needed

Problem 4 Raymond's tree-based DME algorithm [25pts]

Recall the Raymond's tree-based algorithm for distributed mutual exclusion, and answer the following question. Can this algorithm guarantee that all requests are serviced in the FCFS order? If yes, provide your justification. If not, provide an example to show that requests may not be serviced in the FCFS order. Here, let us assume the message transition time is 0 (i.e., a message can be sent instantly between its sender and receiver, and so message passing delay would not be considered as a factor to violate FCFS); in case that several requests are raised at the same time, the execution of them in any order within a row is not considered as a violation of FCFS either because tie is allowed to be broken in any way.

- We claim that the algorithm cannot guarantee FCFS and provide a counterexample to prove it
- Setup
    - Consider a system with 4 sites: S1, S2, S3, S4
    - Tree structure of the sites

```
graph BT
    S4((S4)) --> S2((S2))
    S2((S2)) --> S1((S1))
    S3((S3)) --> S1((S1))
```

    - We also know that the message transition time is 0
- Process
    - Making the requests
        - S4 sends a request at t = 0
            - S4 requests the CS
            - S4 adds itself to its request_q and sends REQUEST to S2
            - S2 receives the request, places S4 in its request_q, and sends REQUEST to S1
            - S1 receives the request and places S2 in its request_q
            - Relevant Queues: we have Q1: [S2], Q2: [S4]
        - S3 sends a request at t = 1
            - S3 requests CS
            - S4 adds itself to its request_q and sends REQUEST to S1
            - S1 receives the request and plces S3 in its request_q
            - Relevant Queues: we have Q1: [S2, S3]
        - S2 sends a request at t = 2
            - S2 requests CS
            - S2 places itself in its request_q
            - S2 checks whether its has already sent a request to S1
            - Since it already sent one on behalf of S4, and hasn't received the token yet, S2 doesn't send a new REQUEST to S1
            - Relevant Queues: we have Q1: [S2, S3]
    - Now we look at the sequence after the token starts moving

        - S1 sends the token to site at the top of its queue: S2
            - S1 now points holder to S2
            - S1 sends REQUEST to S2, because S1 still has S3 in its queue
        - S2 receives the token
            - It processes S4 from the top of its queue
        - S4 enters CS, completing the first request
        - S4 exits and sends the token back to S2
        - S2 receives the token
            - It processes the next item in its queue
            - Since S4 sent the token back, S2 processes itself from the queue
        - S2 enters CS, completing the third request
        - S2 exits and sends the token to S1 (its queue is empty now)
        - S1 receives the token
            - It processes the next item in its queue: S3
        - S3 enters CS, completing the second request
    - So we see that in this example the request order was: S4, S4, S2 but the service order was: S4, S2, S3
    - Based on this example we can claim that FCFS is not guaranteed