**ECE428 HW2**

Q1

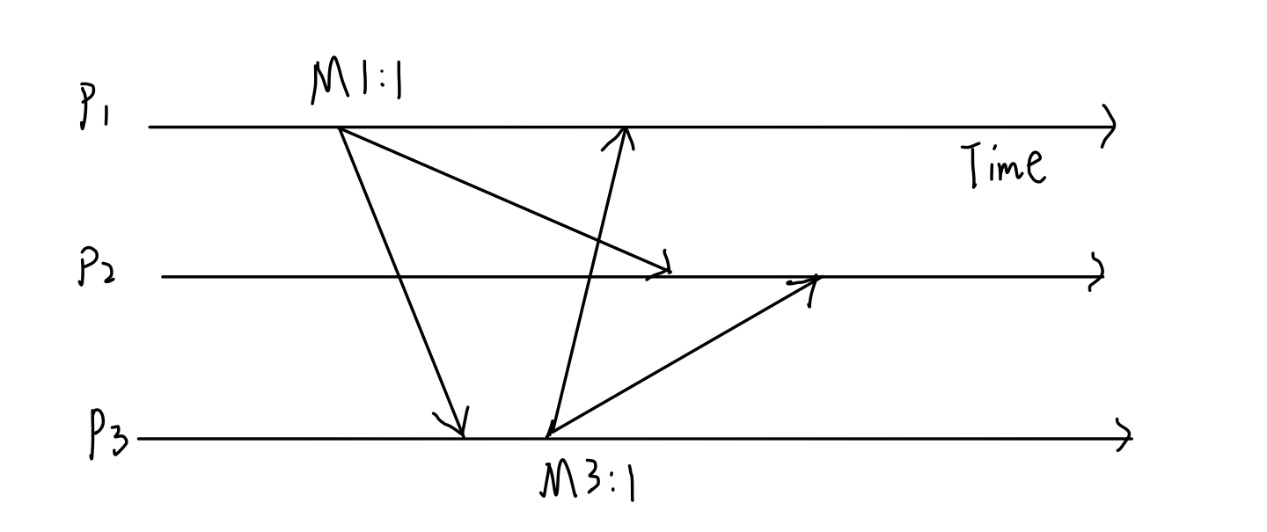
1. It should be FIFO. If process pi multicasts M and then M’, then for every process pj, M will be delieved on the channel from pi to pj before M’. Neither causal ordering nor total ordering should be satisfied. The counter-example works as follows:

Figure 1. counter-example Diagram for Q1.a)

As the figure shows, M1:1 -> M3:1 while p2 deliver M3:1 before M:1:1.

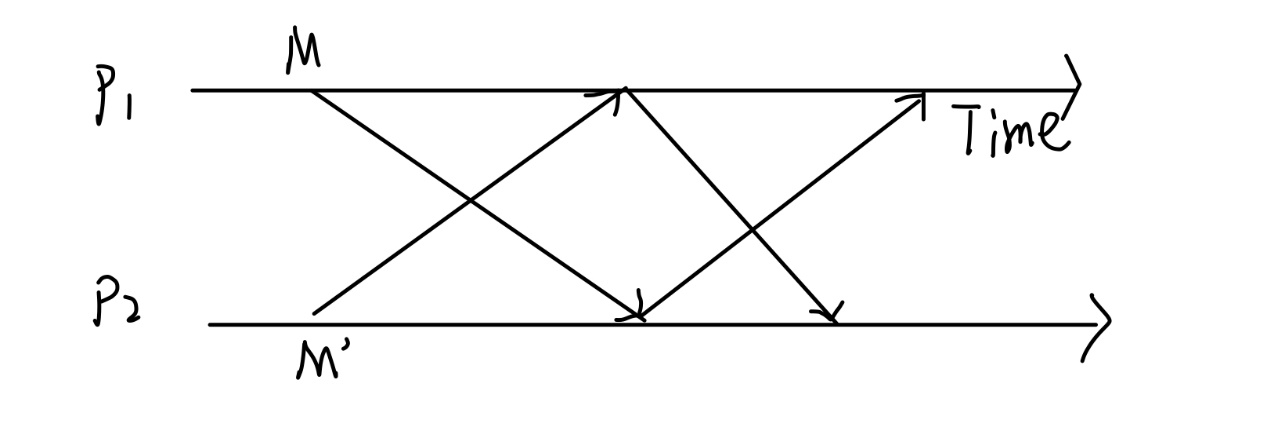
1. It should be causal ordering. When process pi sends message s to pj, any message that has been delivered by pi before s will have already been sent from pi to pj. Total ordering should not satisfy the requirement. The counter-example works as follows:

Figure 2. counter-example Diagram for Q1.b)

As the figure shows, since messages will be remulticasted again for the first time after it is delivered in R-multicast while the total ordering would delay it.

1. R-multicast is causal but not total, because the system with 2 processes fits the requirements while with more processes it does not work/
2. The causal ordering satisfies while total ordering does not. As the algorithm could satisfies that the final priority of M’ is larger than the final priority of M under the condition that M is initially sent before M’ is sent. The total ordering does not work.

Q2

1. To ensure the FIFO multicast delivery order, the messages that have to be delayed in a holdback buffer are: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16. The earliest point at which they can be delivered is when their sequence number is equal to the lowest missing sequence number in the holdback buffer. For example, message 6 can be delivered when all messages 0, 1, 2, 3, 4, 5 have been delivered.
2. To ensure the causal multicast delivery order, messages that have to be delayed are: 5, 6, 7, 8, 9, 10, 11, 12, 13, and 14. The messages are those that have a causal dependency on messages that have not yet been delivered. For example, message 5 has a dependency on message 2, which has not yet been delivered, so message 5 has to be delayed.

|  |  |  |  |
| --- | --- | --- | --- |
| Message | Process | Proposed Priority | Final Priority |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | A  B  C  D  E  C  C  C  C  E  E  D  D  D  E  E  E  NONE | 1  1  1  1  1  2  3  4  5  1  2  2  3  4  3  4  5  NONE | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 |

As the table above shows, the proposed priority of each message is initially set to 1 for each process. In ISIS total ordering, each process sets its own proposed priority for each message that multicasts. The final priority of each message is determined by the ISIS total ordering algorithm, which takes into account the proposed priorities of all processes that have multicast the message. Since no other messages have been seen, the final priorities are simply the order in which the messages are multicast, with ties broken arbitrarily. For example, message 1 has a proposed priority of 1 from process A, so its final priority is also 1. Similarly, message 2 has a proposed priority of 1 from process B, so its final priority is 2, since message 1 was multicast by process A with a proposed priority of 1.

Q3

1. 100

For an R-multicast algorithm, each message is sent to every member of the multicast group, including the sender. Therefore, in a multicast group of 100 nodes, 100 copies of the message will be sent at each multicast.

b) With the modification, the message is only sent to higher-numbered processes. In a multicast group of 100 nodes, if a process has an ID of i, it will send the message to the 99 - i higher-numbered processes. Therefore, if a process has an ID of 1, it will send the message to 98 processes, and if a process has an ID of 99, it will not send the message to any other process.

Hence, the number of messages sent using this modification at each multicast can be calculated as follows:

Number of messages = (99 + 98 + 97 + ... + 1) = (1+99)\*99/2 = 4950

c) def r\_multicast(message, group):

ack\_set = set() # define a acknowledgment set

for member in group:

unicast(member, message)

ack\_set.add(member.id)

while len(ack\_set) > 0:

message\_ack, sender\_id = receive\_ack()

# making the sender waiting

if message\_ack == message and sender\_id in ack\_set:

ack\_set.remove(sender\_id)

for member in group:

deliver(member.id, message)

Q4

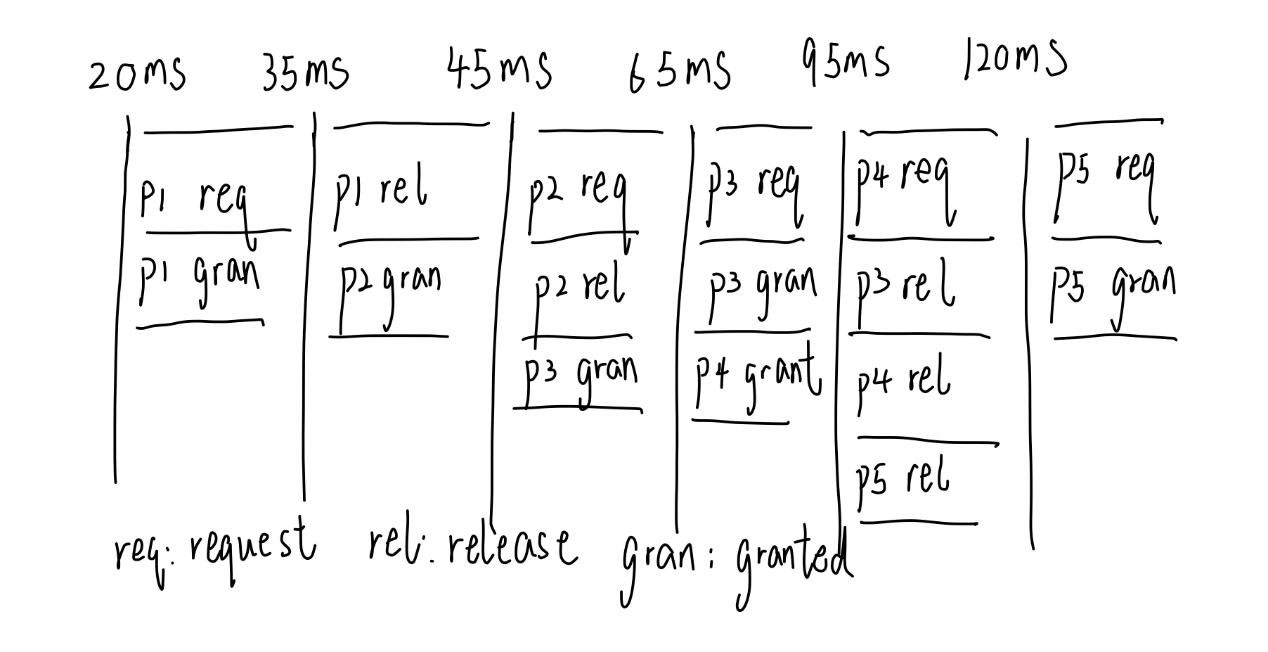
1. **P1 at 20ms, P2 at 25ms, P3 at 30ms, P4 at 55ms, and P5 at 85ms**

Figure 4. e Diagram for timeline of events in Q4.a)

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Time critical section is requested | Time spent in critical section | Time critical section is released |
| P1  P2  P3  P4  P5 | 20ms  43ms  58ms  83ms  122ms | 15ms  10ms  20ms  30ms  25ms | 35ms  45ms  65ms  95ms  120ms |

1. P1 is the leader, it will make the first request to enter the critical section at a time of 20ms. Assuming negligible delay for self-messages, it will be granted access immediately, at a time of 20ms. P1 will spend 15ms in the critical section, so it will release it at a time of 35ms.
2. P2 will request access to the critical section at time 43ms (15ms after P1 releases it). However, there is an 8ms delay between P1 and P2, so P2's request will only reach the central server at time 51ms. P2 will be granted access immediately after P1 releases it, at time 35ms. P2 will spend 10ms in the critical section, so it will release it at time 45ms.
3. P3 will request access to the critical section at time 58ms (15ms after P2 releases it), but its request will only reach the central server at time 66ms, due to the 8ms delay. P3 will be granted access immediately after P2 releases it, at a time of 45ms. P3 will spend 20ms in the critical section, so it will release it at time 65ms.
4. P4 will request access to the critical section at time 83ms (18ms after P3 releases it), but its request will only reach the central server at time 91ms, due to the 8ms delay. P4 will be granted access immediately after P3 releases it, at time 65ms. P4 will spend 30ms in the critical section, so it will release it at time 95ms.
5. P5 will request access to the critical section at time 122ms (27ms after P4 releases it), but its request will only reach the central server at time 130ms, due to the 8ms delay. P5 will be granted access immediately after P4 releases it, at a time of 95ms. P5 will spend 25ms in the critical section, so it will release it at time 120ms.

b) **P1 at 20ms, P4 at 53ms, P2 at 58ms, P3 at 76ms, and P5 at 108ms**

|  |  |  |  |
| --- | --- | --- | --- |
| Process | Time critical section is requested | Time spent in critical section | Time critical section is released |
| P1  P4  P2  P3  P5 | 20ms  55ms  58ms  76ms  108ms | 15ms  30ms  10ms  20ms  25ms | 20ms  83ms  66ms  92ms  133ms |

1. P1 will immediately request access to the critical section at time 0 ms. Assuming negligible delay for self-messages, P1 will be granted access immediately and will spend 15ms in the critical section. P1 will release the token at time 15ms, which means that the token will be passed to P4 at time 23ms (15ms + 8ms delay).
2. P4 will receive the token at time 31ms (23ms + 8ms delay) and will immediately request access to the critical section. Assuming negligible delay for self-messages, P4 will be granted access immediately and will spend 30ms in the critical section. P4 will release the token at time 61ms, which means that the token will be passed to P2 at time 69ms (61ms + 8ms delay).
3. P3 will receive the token at time 103ms (95ms + 8ms delay) and will immediately request access to the critical section. Assuming negligible delay for self-messages, P3 will be granted access immediately and will spend 20ms in the critical section. P3 will release the token at time 123ms, which means that the token will be passed to P5 at time 131ms (123ms + 8ms delay).
4. P5 will receive the token at time 139ms (131ms + 8ms delay) and will immediately request access to the critical section. Assuming negligible delay for self-messages, P5 will be granted access immediately and will spend 25ms in the critical section. P5 will release the token at time 164ms, which means that the token will be passed back to P1 at time 172ms (164ms + 8ms delay).
5. Finally, P1 will receive the token at time 180ms (172ms + 8ms delay) and will immediately request access to the critical section. Assuming negligible delay for self-messages, P1 will be granted access immediately and will spend 15ms in the critical section. P1 will release the token at time 195ms, which means that the token will be passed back to P4 at time 203ms (195ms + 8ms delay).

c) **P1 at 20ms, P4 at 38ms, P2 at 41ms, P3 at 46ms, and P5 at 71ms.**

1. at time 0 ms, all processes request access to the critical section by sending a request message to all other processes with their current lamport timestamp.
2. p1 receives request messages from all other processes, and since it has the lowest timestamp (0), it grants itself permission to enter the critical section at time 0 ms. p1 sends a reply message to all other processes with its current timestamp.
3. p4 receives the reply message from p1 at time 8 ms, and since it has not requested access to the critical section, it sends a reply message to p1 with its current timestamp (1).
4. p2 receives the reply message from p1 at time 16 ms, and since it has not requested access to the critical section, it sends a reply message to p1 with its current timestamp.
5. p3 receives the reply message from p1 at time 24 ms, and since it has not requested access to the critical section, it sends a reply message to p1 with its current timestamp.
6. p5 receives the reply message from p1 at time 32 ms, and since it has not requested access to the critical section, it sends a reply message to p1 with its current timestamp.
7. at time 40 ms, p1 releases the critical section and removes itself from the queue of requests. p1 sends release messages to all other processes with its current timestamp.
8. p4 receives the release message from p1 at time 48 ms, and since it has the next lowest timestamp (1), it grants itself permission to enter the critical section at time 48 ms. p4 sends reply messages to all other processes with its current timestamp .
9. p2 receives the reply message from p4 at time 56 ms, and since it has not requested access to the critical section, it sends a reply message to p4 with its current timestamp.
10. p3 receives the reply message from p4 at time 64 ms, and since it has not requested access to the critical section, it sends a reply message to p4 with its current timestamp.
11. p5 receives the reply message from p4 at time 72 ms, and since it has not requested access to the critical section, it sends a reply message to p4 with its current timestamp.
12. at time 78 ms, p4 releases the critical section and removes itself from the queue of requests. p4 sends release messages to all other processes with its current timestamp.
13. p2 receives the release message from p4 at time 86 ms, and since it has the next lowest timestamp (2), it grants itself permission to enter the critical section at time 86 ms. p2 sends reply messages to all other processes with its current timestamp.
14. p3 receives the reply message from p2 at time 94 ms.