Sanjana Asrani
D17B/01/Batch B

DC04
Lamport's Algorithm

AIM:

Implement a non taken based distributed mutual exclusion. Demonstrate the message overhead complexity by increasing the no of nodes of communication

THEORY:

In distributed computing, mutual exclusion refers to the property where only one process at a time can access a shared resource or critical section, ensuring that conflicting operations do not occur simultaneously. Achieving mutual exclusion in a distributed system poses unique challenges compared to centralized systems due to the lack of shared memory and the presence of independent processes running on separate nodes. Distributed mutual exclusion algorithms address these challenges by coordinating access to shared resources among multiple processes across a network.

Token-based mutual exclusion algorithms rely on the concept of a token, which is a special marker or permission passed among processes to control access to a critical section. These algorithms are designed to ensure that only the process holding the token can enter the critical section, thereby achieving mutual exclusion in a distributed

environment. Token-based algorithms typically involve processes communicating directly with each other to request, pass, and release the token as needed.

1. Suzuki-Kasami Algorithm:

The Suzuki-Kasami algorithm, proposed by K. Suzuki and M. Kasami, is a token-based mutual exclusion algorithm designed for distributed systems. In this algorithm, processes are organized into a logical ring, and a token circulates among them. A process holding the token is granted access to the critical section. Once a process finishes its critical section, it passes the token to the next process in the ring. This algorithm ensures that only one process can enter the critical section at a time, providing mutual exclusion while allowing concurrent execution in non-critical sections.

2. Singhal's Algorithm:

Singhal's algorithm, proposed by Neeraj Kumar Singhal, is a token-based mutual exclusion algorithm similar to the Suzuki-Kasami algorithm. It also uses a logical ring structure for process coordination. However, Singhal's algorithm introduces optimizations to reduce message overhead and improve efficiency compared to Suzuki-Kasami. These optimizations include maintaining additional information about process states and using local knowledge to minimize unnecessary message passing. Singhal's algorithm aims to achieve mutual exclusion with lower communication overhead, making it suitable for large-scale distributed systems.

3. Raymond's Algorithm:

In Raymond's algorithm, processes are organized in a logical ring, and a token circulates among them.
Only the process holding the token can enter the critical section, and after exiting, it passes the token to the next process in the ring.

Non-token-based mutual exclusion algorithms do not rely on the concept of a token for controlling access to critical sections. Instead, these algorithms use various mechanisms such as message passing, logical clocks, or centralized coordination to achieve mutual exclusion among processes in a distributed system. Unlike token-based algorithms, non-token-based approaches may involve more complex message exchange patterns and coordination mechanisms to ensure correct synchronization and access control. These algorithms are often designed to minimize message overhead and ensure fairness and efficiency in resource allocation.

1. Ricart-Agrawala Algorithm:

This algorithm uses a token-based approach where processes communicate directly with each other.
Processes must possess a token to enter the critical section.
When a process requests access, it sends a request message to all other processes and waits for acknowledgments from all of them before entering.

2. Maekawa's Algorithm:

Maekawa's algorithm groups processes into sets and grants access based on voting within these sets.
Each process belongs to multiple sets, and access is granted if a majority of processes in each set approve the request.

3. Lamport's Distributed Mutual Exclusion Algorithm:

This algorithm uses logical timestamps to order events in a distributed system.
Processes broadcast request messages indicating their intention to enter the critical section along with their timestamp.
Access is granted based on the timestamps of incoming requests, ensuring that the process with the lowest timestamp enters first.

Lamport was the first to give a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme. Let Ri be the request set of site Si , i.e. the set of sites from which Si needs permission when it wants to enter CS. In Lamport's algorithm, $\forall i : 1 \leq i \leq N :: Ri= \{S1, S2,...,SN\}$. Every site Si keeps a queue, request_queuei, which contains mutual exclusion requests ordered by their timestamps. This algorithm requires messages to be delivered in the FIFO order between every pair of sites.

Algorithm for Lamport:

Requesting the critical section.

1. When a site Si wants to enter the CS, it sends a REQUEST(tsi, i) message to all the sites in its request set Ri and places the request on request_queuei (tsi is the timestamp of the request).

2. When a site Sj receives the REQUEST(tsi, i) message from site Si, it returns a timestamped REPLY message to Si and places site S'is request on request_queuej.

Executing the critical section

1. Site Si enters the CS when the two following conditions hold:

a) [L1:] Si has received a message with a timestamp larger than (tsi, i) from all other sites.

b) [L2:] S'is request is at the top request_queuei.

Releasing the critical section.

1. Site Si, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.

2. When a site Sj receives a RELEASE message from site Si, it removes S'is request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter CS. The algorithm executes CS requests in the increasing order of timestamps.

Message Complexity

This algorithm creates 3(N − 1) messages per request, or (N − 1) messages and 2 broadcasts. 3(N − 1) messages per request includes:

- (N − 1) total number of requests
- (N − 1) total number of replies
- (N − 1) total number of releases

Performance Analysis : 3(N-1)

Drawbacks:

- Single point of failure is replaced by multiple points of failure.
- message complexity is very high

CODE:

```
import time
def sleep(ms):
    time.sleep(ms / 1000)
def countdown_timer():
    for i in range(5, 0, -1):
        print(i)
        time.sleep(1)
if __name__ == "__main__":
    count_req = 0
    count_res = 0
    count_ok = 0  # reply msgs
    print("Enter number of Processes: ")
    n = int(input())
    interested_processes = []
    sort = [100] * n  # Initialize sort list with 100
    logical_time = 1
    for i in range(n):
        print(f"Does P{i + 1} want to enter the critical region? (yes/no)")
        choice = input().lower()
        if choice == "yes":
            interested_processes.append(i)
            while True:
                print(f"Enter non-negative timestamp for P{i + 1}: ")
                timestamp = int(input())
                if timestamp >= 0:
                    sort[i] = timestamp
                    break
                else:
                    print("Invalid timestamp! Please enter a non-negative integer.")
        else:
            sort[i] = 100

    print("\nTimestamps of processes who need the critical region:")
    for i in range(n):
        print(f"P{i + 1}: {sort[i] if i in interested_processes else '100'}")
    print()
    iterator = 0
    msg_overhead = 0
    start_time = time.time()
    while min(sort) != 100:
        min_val = min(sort)
        min_index = sort.index(min(sort))
        if iterator == 0:
            for i in range(n):
                if sort[i] != 100:
                    for j in range(n):
                        if i != j:
                            if sort[i] < sort[j]:
                                print(f"P{i + 1} -> REQ -> P{j + 1}")
                                count_req += 1
                            elif sort[i] == sort[j]:
                                print("Equal timestamps")
                                print("Decision is being made based on process ID")
                            else:
                                print(f"P{i + 1} -> REQ -> P{j + 1} (Invalid)")
                    count_req += 1
            for i in range(n):
                for j in range(n):
                    if j != min_index and i != j and sort[i] != 100:
                        print("    P" + str(i + 1) + " <- OK <- P" + str(j + 1))
                        count_ok += 1
                    if j == min_index and i != j and sort[i] != 100:
                        count_res += 1
        print("P" + str(min_index + 1) + " gets access to CR")
        countdown_timer()
        print(f"P{min_index + 1} has finished using the critical section")
        print("Identifying the next eligible process...")
        min_val = min(sort)
        min_index = sort.index(min(sort))
        sort[min_index] = 100
        next_min_val = min(sort)
        next_min_index = sort.index(min(sort))
        if next_min_val == 100:
            print("No more processes waiting for the critical section.")
        else:
            print(f"Next eligible process is P{next_min_index + 1} with timestamp {next_min_val}")
        print("")
        iterator += 1
        msg_overhead = count_req + count_res + count_ok
    end_time = time.time()
    print("Total number of request messages : ", count_req)
    print("Total number of reply messages : ", count_ok)
    print("Total number of release messages : ", count_res)
    print("Total message overhead : ", msg_overhead)
    print("Execution time = ", end_time - start_time)
```

OUTPUT:

Taking timestamps as inputs:

```
PS C:\Users\Admin1\Desktop\d17b1> python LamportsAlgo.py
Enter number of Processes:
4
Does P1 want to enter the critical region? (yes/no)
yes
Enter non-negative timestamp for P1:
0
Does P2 want to enter the critical region? (yes/no)
yes
Enter non-negative timestamp for P2:
0
Does P3 want to enter the critical region? (yes/no)
yes
Enter non-negative timestamp for P3:
29
Does P4 want to enter the critical region? (yes/no)
yes
Enter non-negative timestamp for P4:
10
```

If in case, 2 processes have same timestamps: decision is taken based on the process ids

```
Timestamps of processes who need the critical region:
P1: 0
P2: 0
P3: 29
P4: 10

Equal timestamps
Decision is being made based on process ID
P1 -> REQ -> P3
P1 -> REQ -> P4
Equal timestamps
Decision is being made based on process ID
```

Requests and reply messages are sent from Requestor Process to Requested Process
Request is only accepted if the requestor has a smaller timestamp than requested.

```
P2 -> REQ -> P3
P2 -> REQ -> P4
P3 -> REQ -> P1 (Invalid)
P3 -> REQ -> P2 (Invalid)
P3 -> REQ -> P4 (Invalid)
P4 -> REQ -> P1 (Invalid)
P4 -> REQ -> P2 (Invalid)
P4 -> REQ -> P3
    P1 <- OK <- P2
    P1 <- OK <- P3
    P1 <- OK <- P4
    P2 <- OK <- P3
    P2 <- OK <- P4
    P3 <- OK <- P2
    P3 <- OK <- P4
    P4 <- OK <- P2
    P4 <- OK <- P3
```

The process goes into C.R. for execution:

```
P1 gets access to CR
5
4
3
2
1
P1 has finished using the critical section
```

Then the next requestor process follows the same process

```
Identifying the next eligible process...
Next eligible process is P2 with timestamp 0

P2 gets access to CR
5
4
3
2
1
P2 has finished using the critical section
Identifying the next eligible process...
Next eligible process is P4 with timestamp 10

P4 gets access to CR
5
4
3
2
1
P4 has finished using the critical section
Identifying the next eligible process...
Next eligible process is P3 with timestamp 29


P3 gets access to CR
5
4
3
2
1
P3 has finished using the critical section
Identifying the next eligible process...
No more processes waiting for the critical section.

Total number of request messages :  9
Total number of reply messages :  9
Total number of release messages :  3
Total message overhead :  21
Execution time =  20.024388074874878
PS C:\Users\Admin1\Desktop\d17b1>
```

Conclusion:

Thus the non token based algorithm: **Lamport's** Algorithm has been successfully implemented where the younger process i.e. the one with smaller timestamp is allowed an entry to C.S. (Critical Section). Verification of number of message overheads being 3n-1 where n is the number of processes has been done using the sums of request messages, reply messages, and release messages between the processes, N – 1 REQUEST messages to all process (N minus itself), N −1 REPLY messages, and N −1 RELEASE messages per CR invocation has been done. The synchronization delay is T. Throughput is 1/(T + E). The algorithm has been proven to be fair and correct. It can also be optimized by reducing the number of RELEASE messages sent