20CYS402 – Distributed Systems and Cloud Computing

Lab Exercise – 3

Name: R Subramanian

Roll Number: CH.EN.U4CYS22043

Objective

To understand and implement **mutual exclusion** in distributed systems using:

1. **Ricart–Agrawala Algorithm** (Timestamp Prioritized Scheme)
2. **Maekawa's Algorithm** (Voting Scheme)

1. Timestamp Prioritized Scheme (Ricart–Agrawala Algorithm)

Algorithm Steps:

1. A process sends a **REQUEST(timestamp, process_id)** to all other processes.
2. Upon receiving a REQUEST:
   - If the receiver is not interested or has a **higher timestamp**, it sends a **REPLY** immediately.
   - Otherwise, it **queues the request**.
3. A process enters the **critical section** after receiving **REPLY** from all other processes.
4. Upon exiting the critical section, the process sends **REPLY** to all queued requests.

Program Code:

```python
class Process:
    def __init__(self, pid):
        self.pid = pid
        self.timestamp = 0
        self.request_queue = []
        self.replies_needed = 2

    def request_cs(self, other_processes):
        self.timestamp += 1
        print(f"\nProcess {self.pid} requesting CS with timestamp {self.timestamp}")
        replies = 0

        for proc in other_processes:
            if proc.timestamp == 0 or (proc.timestamp > self.timestamp) or (proc.timestamp == self.timestamp and proc.pid > self.pid):
                print(f"Process {proc.pid} sends REPLY to {self.pid}")
                replies += 1
            else:
                print(f"Process {proc.pid} queues request from {self.pid}")

        if replies == self.replies_needed:
            self.enter_cs()

    def enter_cs(self):
        print(f"{self.pid} enters Critical Section")
        print(f"{self.pid} exits Critical Section\n")
        self.timestamp = 0

P0 = Process(0)
P1 = Process(1)
P2 = Process(2)

P0.request_cs([P1, P2])
P1.request_cs([P0, P2])
P2.request_cs([P0, P1])
```

Sample Output:

```
[Running] python -u "c:\Users\amma\Documents\DSCC\Timestamp.py"

Process 0 requesting CS with timestamp 1
Process 1 sends REPLY to 0
Process 2 sends REPLY to 0
0 enters Critical Section
0 exits Critical Section


Process 1 requesting CS with timestamp 1
Process 0 sends REPLY to 1
Process 2 sends REPLY to 1
1 enters Critical Section
1 exits Critical Section


Process 2 requesting CS with timestamp 1
Process 0 sends REPLY to 2
Process 1 sends REPLY to 2
2 enters Critical Section
2 exits Critical Section
```

---

2. Voting Scheme (Maekawa's Algorithm)

At Requesting Process (P):

1. Send **REQUEST** message to all processes in its **voting set**.
2. Wait until **REPLY** is received from all members of the voting set.
3. Enter the **critical section**.
4. Upon exit, send **RELEASE** to all members of the voting set.

At Voting Process (Q):

1. On receiving a **REQUEST**:
   - If not voted yet, send **REPLY** and mark the vote as granted.
   - If already voted, queue the request.
2. On receiving a **RELEASE**:
   - Send **REPLY** to the next request in the queue (if any).
   - Update vote status accordingly.

Program Code:

VotingScheme.py > ...

```python
class Voter:
    def __init__(self, vid):
        self.vid = vid
        self.voted = False

    def vote(self, pid):
        if not self.voted:
            self.voted = True
            print(f"Voter {self.vid} votes for Process {pid}")
            return True
        else:
            print(f"Voter {self.vid} has already voted")
            return False

    def release(self):
        self.voted = False


class Process:
    def __init__(self, pid, voters):
        self.pid = pid
        self.voters = voters

    def request_cs(self):
        print(f"\nProcess {self.pid} requesting CS")
        votes_received = 0
        for v in self.voters:
            if v.vote(self.pid):
                votes_received += 1

        if votes_received == len(self.voters):
            print(f"{self.pid} enters Critical Section")
            print(f"{self.pid} exits Critical Section")
            for v in self.voters:
                v.release()

V0 = Voter(0)
V1 = Voter(1)
V2 = Voter(2)

P0 = Process(0, [V0, V1])
P1 = Process(1, [V1, V2])
P2 = Process(2, [V0, V2])

P0.request_cs()
P1.request_cs()
P2.request_cs()
```

Sample Output:

```
[Running] python -u "c:\Users\amma\Documents\DSCC\VotingScheme.py"

Process 0 requesting CS
Voter 0 votes for Process 0
Voter 1 votes for Process 0
0 enters Critical Section
0 exits Critical Section

Process 1 requesting CS
Voter 1 votes for Process 1
Voter 2 votes for Process 1
1 enters Critical Section
1 exits Critical Section

Process 2 requesting CS
Voter 0 votes for Process 2
Voter 2 votes for Process 2
2 enters Critical Section
2 exits Critical Section
```

Conclusion

Mutual exclusion in distributed systems can be efficiently implemented using either **timestamp-based** schemes or **voting-based** schemes. Proper coordination logic ensures:

- Consistency of critical section access.
- Prevention of simultaneous entry by multiple processes.
- Deadlock avoidance and fair access among processes.
