CEL72, DSL, Monsoon 2020

Lab 4: Implementation of Mutual Exclusion algorithm

Batch A, Student ID:

**Objective**

Implementation of Mutual Exclusion algorithm.

**Introduction**

In computer science, mutual exclusion refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time; it is a basic requirement in concurrency control, to prevent race conditions. Here, a critical section refers to a period when the process accesses a shared resource, such as shared memory. The requirement of mutual exclusion was first identified and solved by Edsger W. Dijkstrain his seminal 1965 paper titled Solution of a problem in concurrent programming control, and is credited as the first topic in the study of concurrent algorithms.[1,2].

**Scenario: Why mutual exclusion?**

Consider Bank’s Servers are in the Cloud:

* Two of your friends make simultaneous deposits of Rs.10000 into your bank account, each from a separate ATM.
* Both ATMs read initial amount of Rs. 1000 Rs. concurrently from the bank’s cloud server.
* Both ATMs add Rs. 10,000 to this amount (locally at the ATM).
* Both write the final amount to the server.
* What’s wrong?
* You lost Rs.10,000!
* The ATMs need mutually exclusive access to your account entry at the
* or, mutually exclusive access to executing the code that modifies the account entry

**Problem Statement**

* Critical Section Problem: Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.
* Each process can call three functions
  + enter() to enter the critical section (CS)
  + AccessResource() to run the critical section code
  + exit() to exit the critical section

**Implementation Procedure:**

**Approaches to solve mutual Exclusion in Distributed System**

* Processes communicating by passing messages
* Need to guarantee 3 properties:
  + Safety (essential) – At most one process executes in CS (Critical Section) at any time
  + Liveness (essential) – Every request for a CS is granted eventually
  + Ordering (desirable) – Requests are granted in the order they were made

**Specify System Model**:

* Each pair of processes is connected by reliable channels (such as TCP).
* Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
* Processes do not fail.

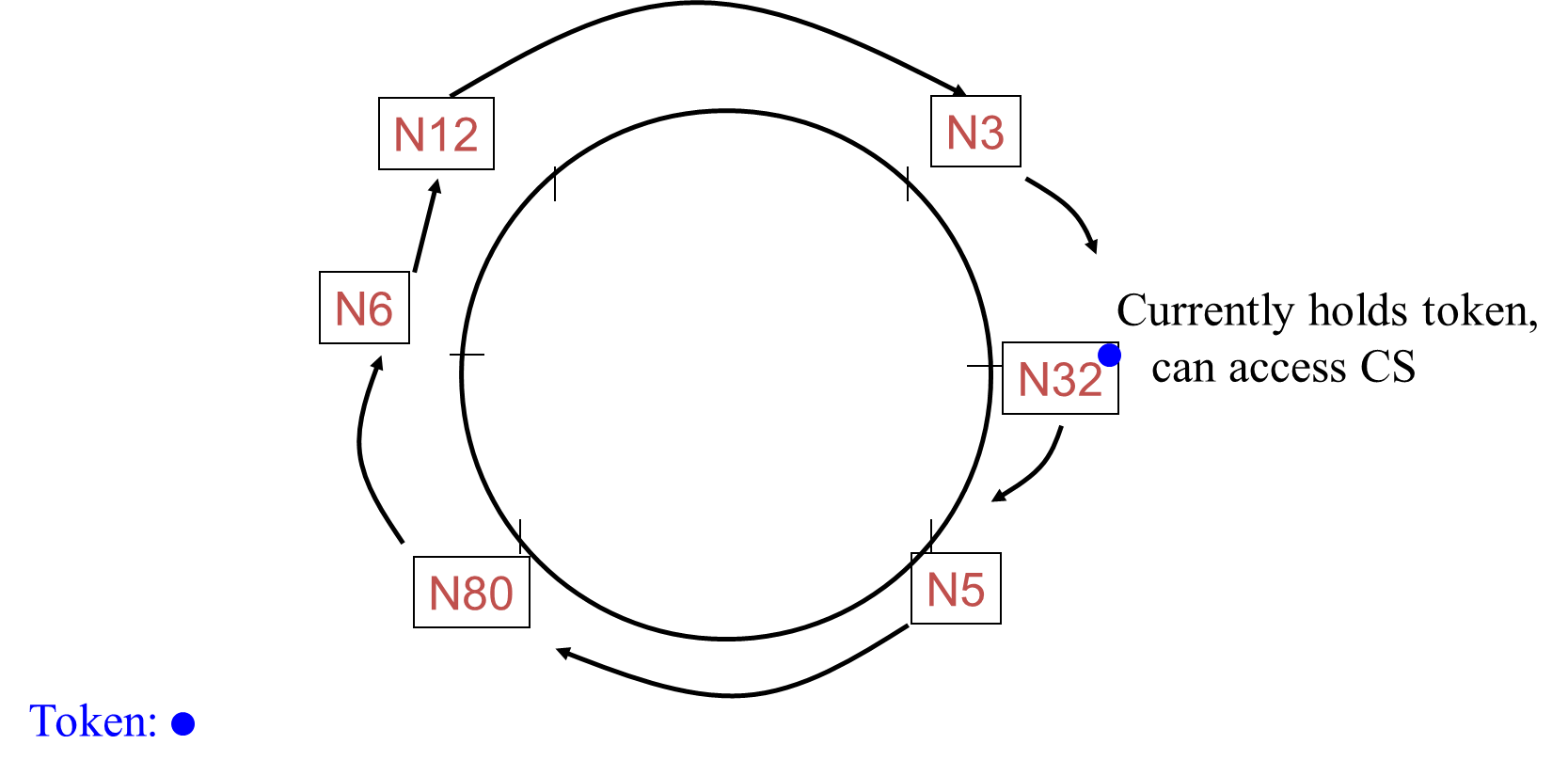
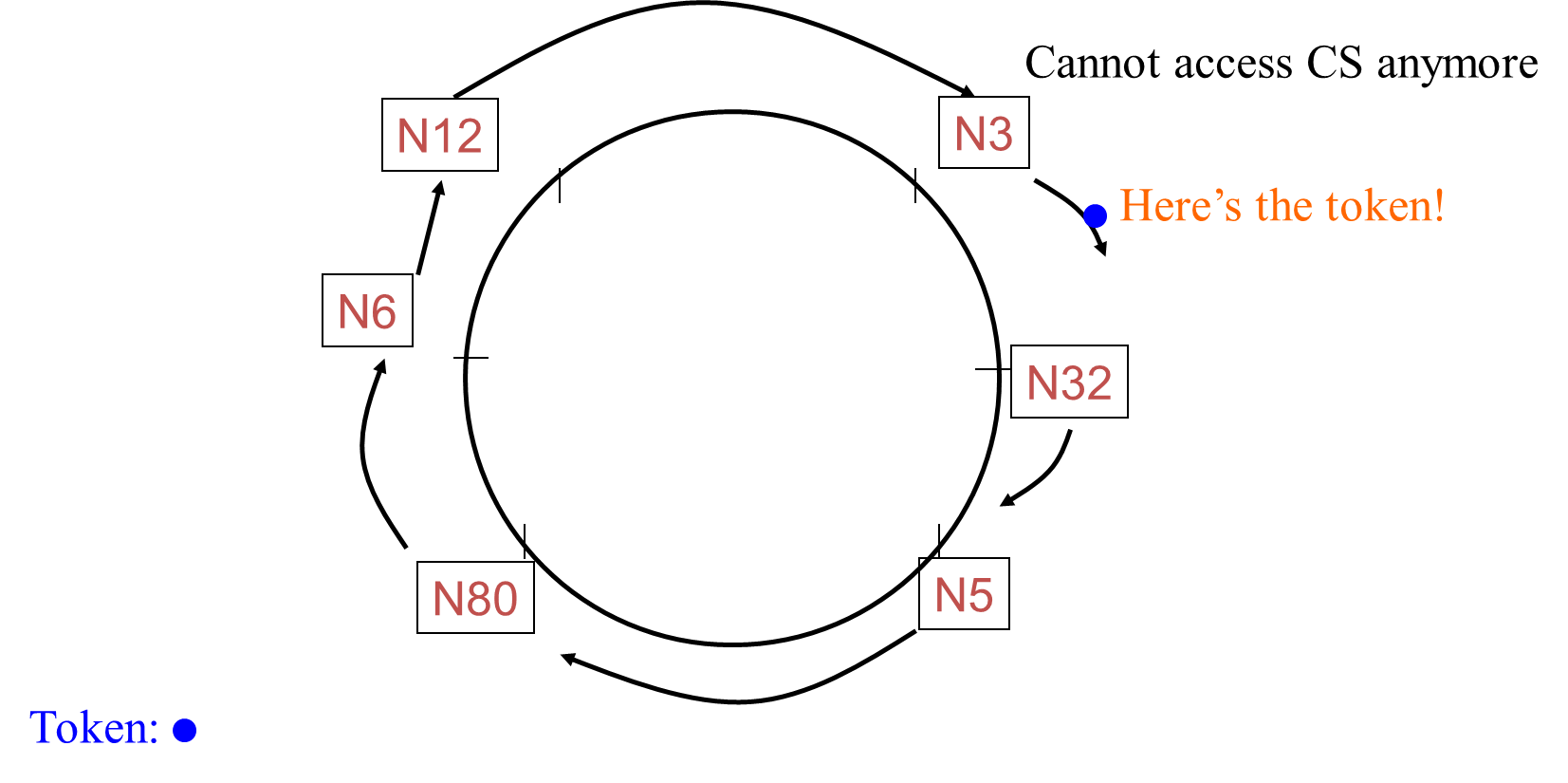
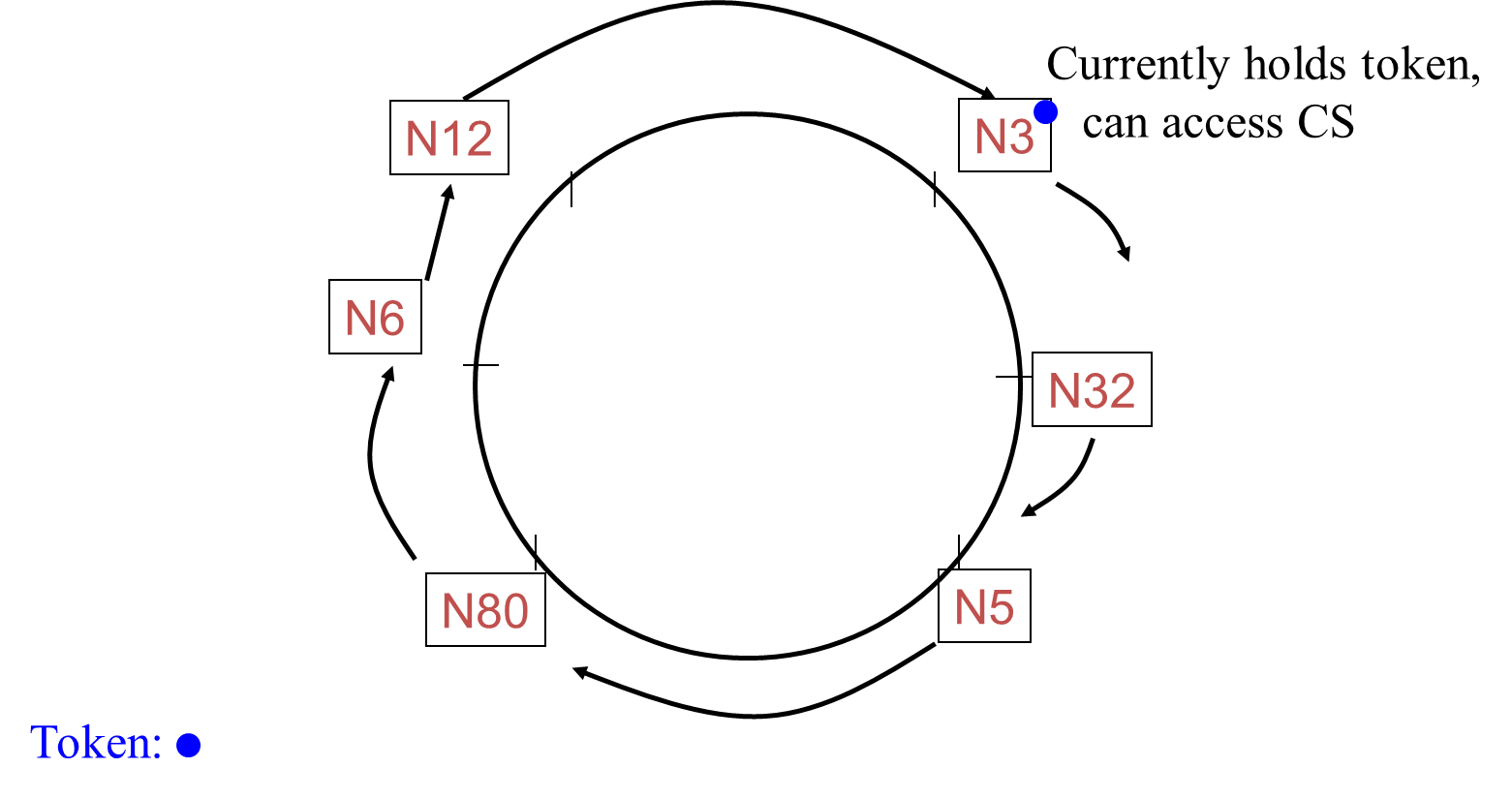
**Algorithms:**

**Central Solution [1]**

1. Elect a central master (or leader)
   1. Use one of our election algorithms!
2. Master keeps
   1. A **queue** of waiting requests from processes who wish to access the CS
   2. A special **token** which allows its holder to access CS
3. Actions of any process in group:
   1. enter()
      1. Send a request to master
      2. Wait for token from master
   2. exit()
      1. Send back token to master

**Ring Based Solution [1]**

1. *N* Processes organized in a virtual ring
2. Each process can send message to its successor in ring
3. Exactly 1 token
4. enter()
   1. Wait until you get token
5. exit() // already have token
   1. Pass on token to ring successor
6. If receive token, and not currently in enter(), just pass on token to ring successor



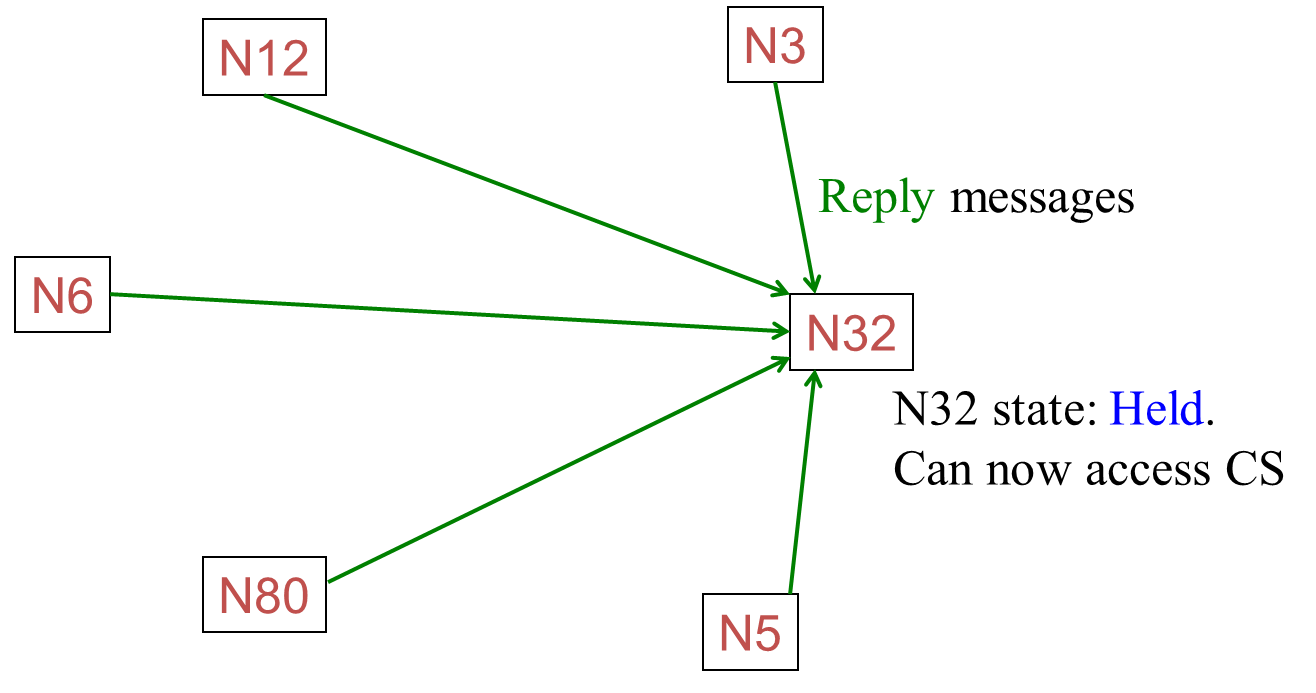
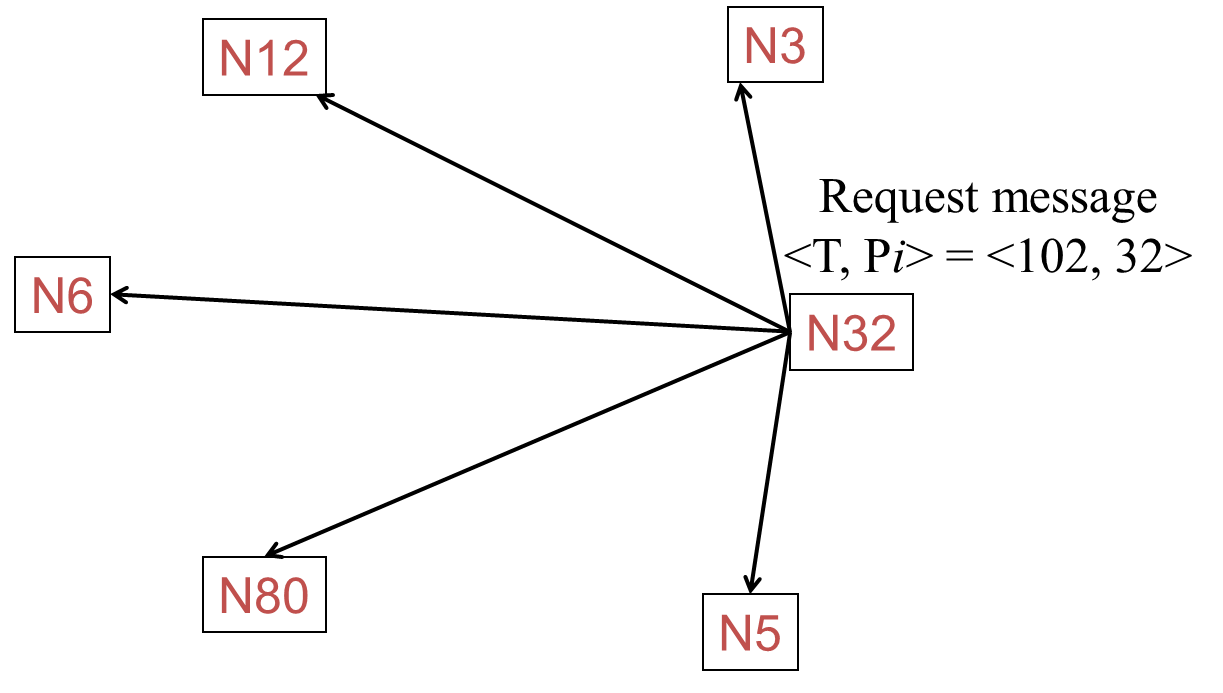
**Ricart-Agrawala Algorithm [1, 2]**

Classical algorithm from 1981, Invented by Glenn Ricart (NIH) and Ashok Agrawala (U. Maryland), No token. Uses the notion of causality and multicast. Has lower waiting time to enter CS than Ring-Based approach

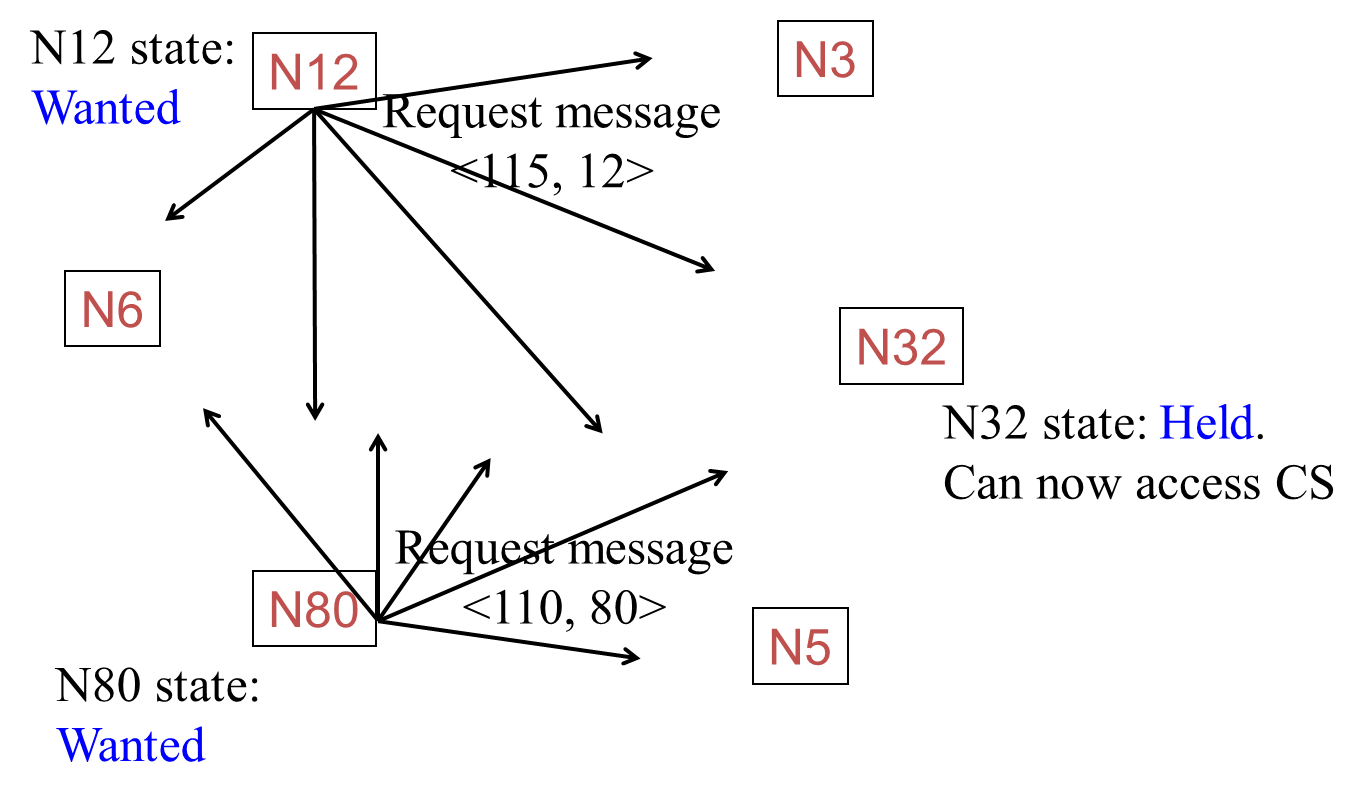
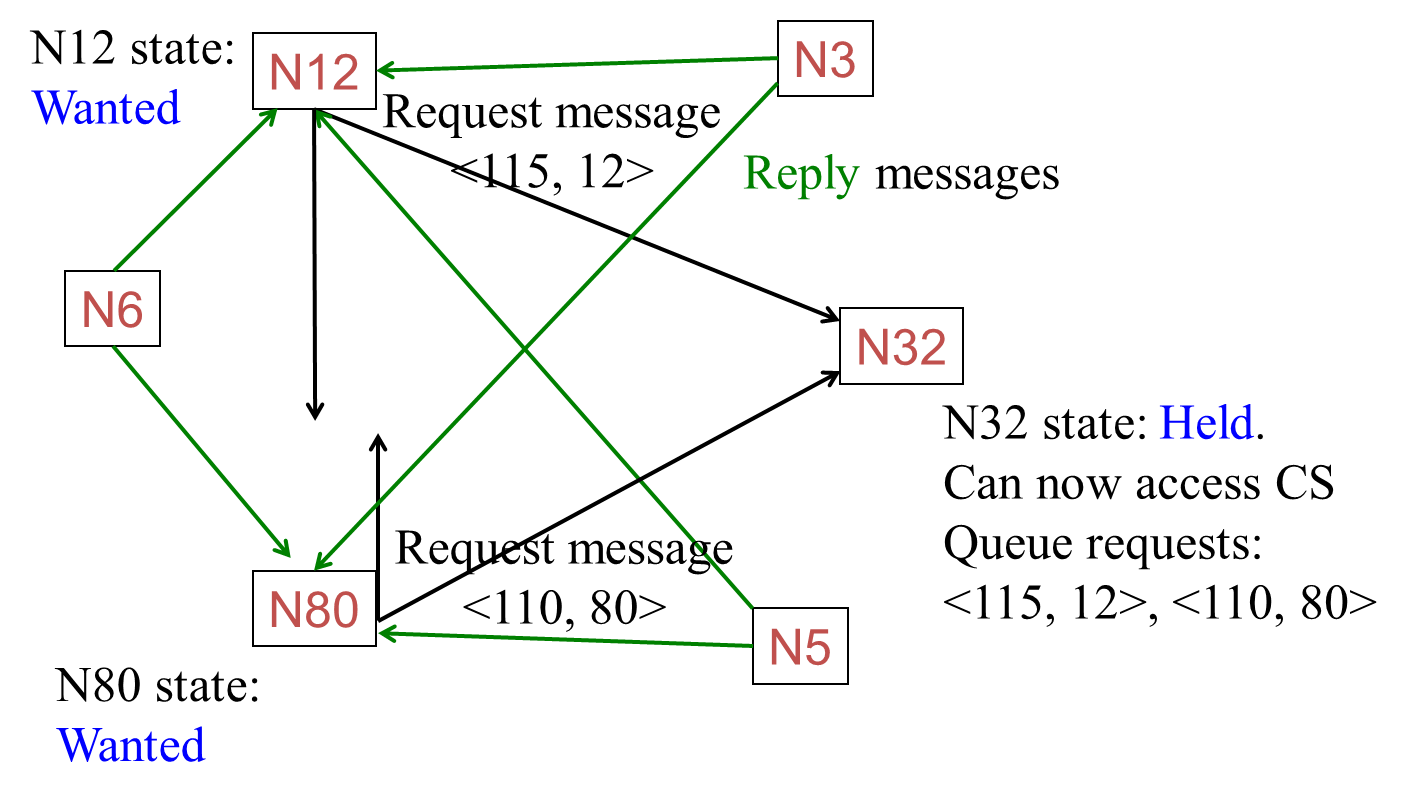
1. enter() at process P*i*
   1. multicast a request to all processes
      1. Request: <T, P*i*>, where T = current Lamport timestamp at P*i*
   2. Wait until *all* other processes have responded positively to request
2. Requests are granted in order of causality
3. <T, P*i*> is used lexicographically: P*i* in request <T, P*i*> is used to break ties (since Lamport timestamps are not unique for concurrent events)

***Messages in RA Algo***

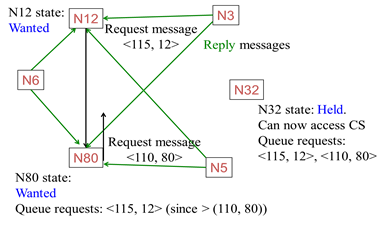
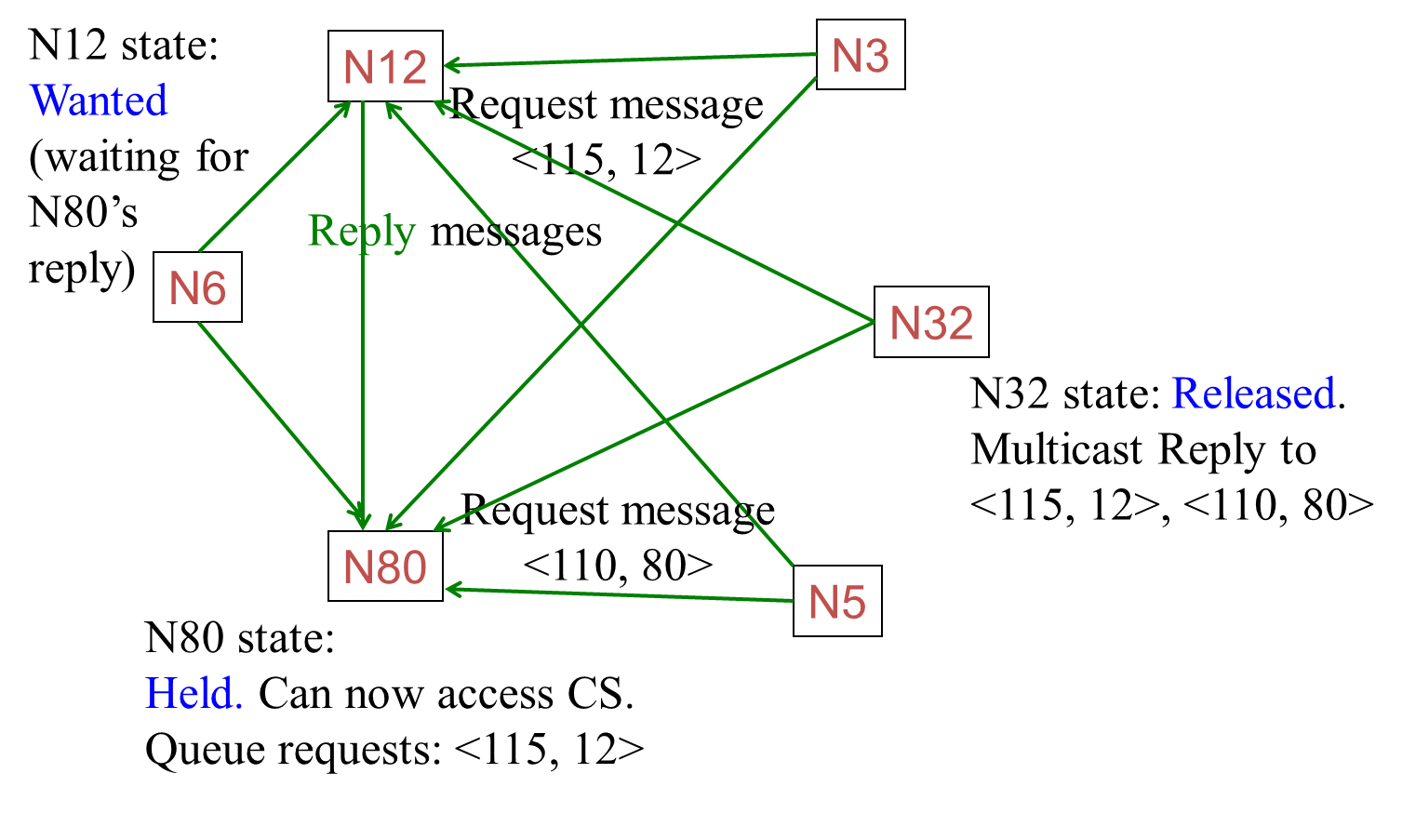
1. enter() at process P*i*
   1. set state to Wanted
   2. multicast “Request” <T*i*, P*i*> to all processes, where T*i* = current Lamport timestamp at P*i*
   3. wait until ***all*** processes send back “Reply”
   4. change state to Held and enter the CS
2. On receipt of a Request <T*j,* P*j*> at P*i* (*i ≠ j*)*:*
   1. **if** (state = Held) or (state = Wanted & (T*i*, *i*) < (T*j*, *j*))
      1. // lexicographic ordering in (T*j*, P*j*)
      2. add request to local queue (of waiting requests)
3. **else** send “Reply” to P*j*
4. exit() at process P*i*
   1. change state to Released and “Reply” to *all* queued requests.



* + - 1. (b)



* + - 1. (d)

(e) (f)

**Answer following questions:**

1. Provide analysis from Safety. Liveness and ordering
2. Explain performance in terms of Bandwidth, Client Delay and synchronization delay

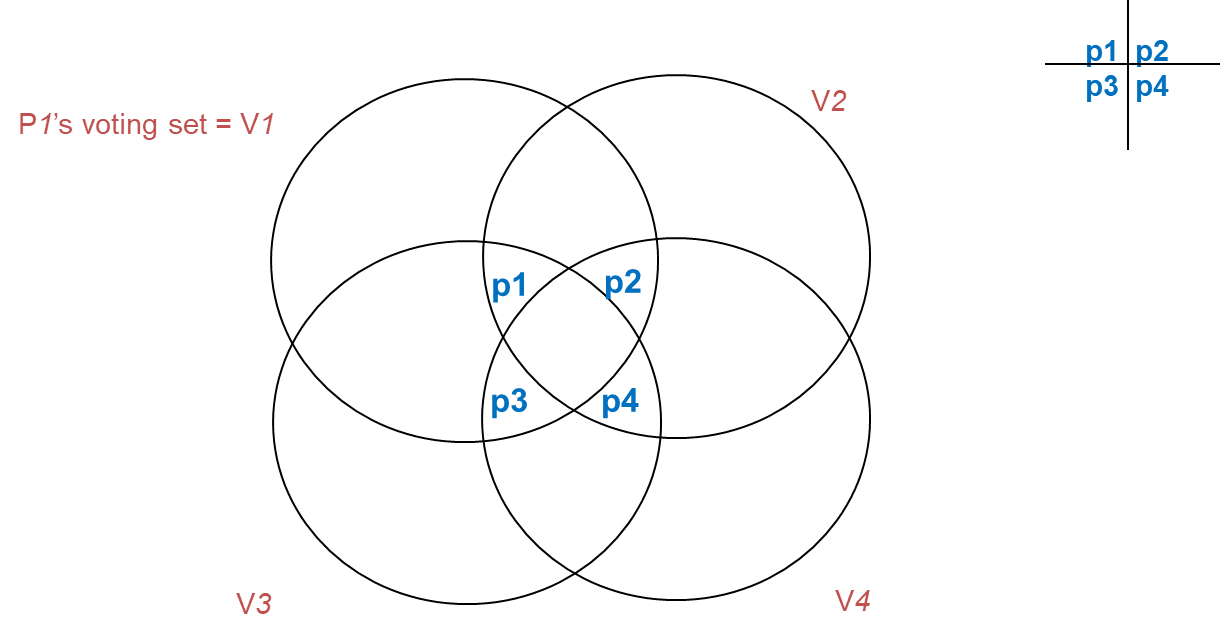
**Maekawa’s Algorithm [1, 2]**

Ricart-Agrawala requires replies from *all* processes in group. In Maekawa algo instead, it get replies from only *some* processes in group .But ensure that only one process is given access to CS (Critical Section) at a time.

***Maekawa Voting sets***

1. Each process P*i* is associated with a *voting set* V*i* (of processes)
2. Each process belongs to its own voting set
3. *The intersection of any two voting sets must be non-empty*
   1. *Same concept as Quorums!*
4. Each voting set is of size *K*
5. Each process belongs to *M* other voting sets
6. Maekawa showed that *K=M=*√*N* works best
7. One way of doing this is to put N processes in a √*N* by √*N* matrix and for each P*i*, its voting set V*i* = row containing P*i* + column containing P*i.* Size of voting set = 2\*√*N*-1

Voting Sets with N+4



* Each process requests permission from only its voting set members
  + Not from all
* Each process (in a voting set) gives permission to at most one process at a time
  + Not to all

***Actions:***

* state = Released, voted = false
* enter() at process P*i*:
  + state = Wanted
  + Multicast Request message to all processes in V*i*
  + Wait for Reply (vote) messages from all processes in V*i* (including vote from self)
  + state = Held
* exit() at process P*i*:
  + state = Released
  + Multicast Release to all processes in V*i*
* When P*i* receives a Request from P*j*:

**if** (state == Held OR voted = true)

queue Request

else

send Reply to P*j* and set voted = true

* When P*i* receives a Release from P*j*:

**if** (queue empty)

voted = false

else

dequeue head of queue, say P*k*

Send Reply *only* to P*k*

voted = true

**Answer following questions:**

1. Provide analysis from Safety. Liveness and ordering

**Safety** is assured as one process can cast only one vote at a time and due to the intersection property of Quorums no to processes can enter the critical sections together.

**Liveliness** is assured by modification to the original Maekawa Algorithm by introducing the Yield and Enquire messages.

**Ordering** is assured by Including time stamps in the request message and the process with the smaller timestamp is treated as a higher priority process and other processes with lesser priorities are forced to yield in case of a request from a higher priority process for CS.

2. Explain performance in terms of Bandwidth, Client Delay and synchronization delay

Since the size of a request set is √N, an execution of the CS requires √N REQUEST, √N REPLY, and √N RELEASE messages, resulting in **3√N** messages per CS execution.

**Synchronization** delay in this algorithm is 2T. This is because after a site Si exits the CS, it first releases all the sites in Ri (Quorum) and then one of those sites sends a REPLY message to the next site that executes the CS.

For avoiding deadlocks additional exchanges of 2 messages are required. Therefore 5√N exchanges are required. (Enquire and Failed messages)

**Conceptual architecture of the distributed system you have considered**

**Processes and their Quorums**

There are total 9 processes and they are divided into quorums as follows:

**QUORUM FOR: 1 [1, 2, 3, 4, 7]**

**QUORUM FOR: 2 [1, 2, 3, 5, 8]**

**QUORUM FOR: 3 [1, 2, 3, 6, 9]**

**QUORUM FOR: 4 [1, 4, 5, 6, 7]**

**QUORUM FOR: 5 [2, 4, 5, 6, 8]**

**QUORUM FOR: 6 [3, 4, 5, 6, 9]**

**QUORUM FOR: 7 [1, 4, 7, 8, 9]**

**QUORUM FOR: 8 [2, 5, 7, 8, 9]**

**QUORUM FOR: 9 [3, 6, 7, 8, 9]**

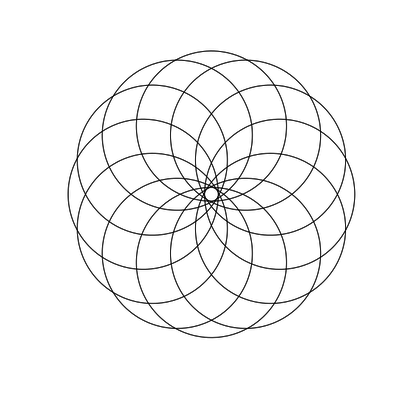


Fig. Maekawa Quorums representation for 12 processes

**Source files**

import math

import sys

import Queue

import threading

from termcolor import colored

import time

import random

import copy

import collections

N = 9

n = int(math.sqrt(N))

assert math.sqrt(N) \*\* 2 == N, "N must be a square number"

assert len(sys.argv) == 5, "Require {} arguments to function, given {}".format(4, len(sys.argv) - 1)

'''

EXECUTING PARAMETERS

'''

try:

cs\_int = int(sys.argv[1]) ## Critical Section (CS) Execution Time

next\_req = int(sys.argv[2]) ## Next Request for CS

tot\_exec\_time = int(sys.argv[3]) ## Total Execution time of each process

option = int(sys.argv[4]) ## Option for printting the logs on to the terminals

except ValueError:

print "Invalid command line arguments provided. Require integers."

sys.exit(1)

def returnColumn(matrix,col):

vals = set()

for i in range(len(matrix)):

for j in range(len(matrix)):

if j == col:

vals.add(matrix[i][j])

return vals

def main():

# Each element of threads is a list with the following indices

# 0: message queue

# 1: dictionary used to track various variables for each

# 2: thread object

global threads

threads = []

threads.append(None)

mat = [[0 for i in range(n)] for i in range(n)]

prcs = 1

for i in range(n):

for j in range(n):

mat[i][j] = prcs

prcs += 1

quorums = collections.defaultdict(set)

for i in range(n):

for j in range(n):

quorums[mat[i][j]] = quorums[mat[i][j]].union(set(mat[i]))

quorums[mat[i][j]] = quorums[mat[i][j]].union(returnColumn(mat,j))

'''

PRINTING QUORUMS

'''

for k,v in quorums.items():

print("QUORUM FOR: " + str(k) + str(list(v)))

# Initialize threads

for x in range(1, N + 1):

a = []

a.append(Queue.PriorityQueue())

a.append({})

a.append(threading.Thread(target=main\_thread\_function, args=(x,)))

threads.append(a)

# Start threads

shuffled = range(1, N + 1)

random.shuffle(shuffled)

for x in shuffled:

threads[x][1]['sem'] = threading.Semaphore()

threads[x][1]['vote'] = None

threads[x][1]['nodes'] = []

threads[x][2].daemon = True

threads[x][2].start()

time.sleep(tot\_exec\_time)

def main\_thread\_function(thread\_id):

threads[thread\_id][1]['state'] = "Idle"

threads[thread\_id][1]['child'] = threading.Thread(

target=message\_handler\_threads, args=(thread\_id,))

threads[thread\_id][1]['child'].daemon = True

threads[thread\_id][1]['child'].start()

while 1:

r\_time = time.time()

while 1:

threads[thread\_id][1]['state'] = "Requesting"

threads[thread\_id][1]['sem'] = threading.Semaphore()

request\_critical(thread\_id, r\_time=r\_time)

wait\_for\_critical(thread\_id)

if threads[thread\_id][1]['state'] is not "Failed":

break

print "{} - Acquired from {}".format(thread\_id, " ".join([str(x) for x in threads[thread\_id][1]['nodes']]))

threads[thread\_id][1]['state'] = "Acquired"

time.sleep(cs\_int / 1000.0)

print "{} - Released".format(thread\_id)

release\_critical(thread\_id, r\_time=r\_time)

threads[thread\_id][1]['state'] = "Idle"

time.sleep(next\_req / 1000.0)

def wait\_for\_critical(thread\_id):

for x in range(0, 2 \* n):

threads[thread\_id][1]['sem'].acquire()

def message\_handler\_threads(thread\_id):

while 1:

time, msg = threads[thread\_id][0].get()

time = copy.copy(time)

msg = copy.copy(msg)

if msg['action'] is "request":

if threads[thread\_id][1]['vote'] is None:

send\_grant\_message(thread\_id, msg)

else:

if threads[thread\_id][1]['vote'][0] > msg['tstamp']:

if threads[thread\_id][1]['vote'][1] == msg['src']:

send\_grant\_message(thread\_id, msg)

else:

send\_inquire\_message(thread\_id, msg)

else:

if threads[thread\_id][1]['vote'][1] == msg['src'] and threads[thread\_id][1]['vote'][0] == msg['tstamp']:

send\_grant\_message(thread\_id, msg)

else:

threads[thread\_id][0].put((msg['tstamp'], msg))

elif msg['action'] is "grant":

if msg['src'] not in threads[thread\_id][1]['nodes']:

threads[thread\_id][1]['nodes'].append(msg['src'])

print "\t{}({:.6f}) - Received {} from {}. Votes: {}\n".format(thread\_id, msg['tstamp'], msg['action'], msg['src'], " ".join([str(x) for x in threads[thread\_id][1]['nodes']])),

threads[thread\_id][1]['sem'].release()

elif msg['action'] is "release":

threads[thread\_id][1]['vote'] = None

elif msg['action'] is "failed":

msg['action'] = 'request'

msg['src'] = msg['alternative']

msg.pop('alternative')

threads[thread\_id][0].put((msg['tstamp'], msg))

elif msg['action'] is "inquire":

state = threads[thread\_id][1]['state']

if (state is "Requesting" or state is "Idle"):

threads[thread\_id][1]['state'] = "Failed"

for x in range(2 \* n):

threads[thread\_id][1]['sem'].release()

send\_relinquish\_message(thread\_id, msg)

else:

send\_failed\_message(thread\_id, msg)

elif msg['action'] is "relinquish":

if threads[thread\_id][1]['vote'] is not None and threads[thread\_id][1]['vote'][1] is msg['src']:

msg['src'] = msg['alternative']

msg['action'] = "grant"

msg.pop('alternative')

send\_grant\_message(thread\_id, msg)

else:

print "\t{} - Old RELINQUISH received from {}. Alternative: {}.\n".format(thread\_id, msg['src'], msg['alternative']),

else:

print colored("Unknown action '{}' received!", "red").format(msg['action'])

def send\_message(dst, msg):

threads[dst][0].put((msg['tstamp'], msg))

# Reply to src.

def send\_grant\_message(thread\_id, imsg):

# print "\t{} sending GRANT to {}\n".format(thread\_id, imsg['src']),

threads[thread\_id][1]['vote'] = (imsg['tstamp'], imsg['src'])

imsg['action'] = 'grant'

dst = imsg['src']

imsg['src'] = thread\_id

send\_message(dst, imsg)

# Reply to src with original message.

def send\_failed\_message(thread\_id, imsg):

# print "\t{} sending FAILED to {}. State: {}\n".format(thread\_id, imsg['src'], threads[thread\_id][1]['state']),

dst = imsg['src']

imsg['src'] = thread\_id

imsg['action'] = 'failed'

send\_message(dst, imsg)

# Reply to current voted node with same message, set alternative.

def send\_inquire\_message(thread\_id, imsg):

# print "\t{} sending INQUIRE to {}. Alternative: {}\n".format(thread\_id, threads[thread\_id][1]['vote'][1], imsg['src']),

imsg['alternative'] = imsg['src']

imsg['src'] = thread\_id

imsg['action'] = 'inquire'

send\_message(threads[thread\_id][1]['vote'][1], imsg)

# Reply to src with same message.

def send\_relinquish\_message(thread\_id, imsg):

dst = imsg['src']

imsg['src'] = thread\_id

imsg['action'] = 'relinquish'

send\_message(dst, imsg)

def request\_critical(thread\_id, r\_time=time.time()):

threads[thread\_id][1]['nodes'] = []

msg = {

"action": "request",

"src": thread\_id,

"tstamp": r\_time

}

send\_to\_voting\_set(thread\_id, msg)

def release\_critical(thread\_id, r\_time=time.time()):

msg = {

"action": "release",

"src": thread\_id,

"tstamp": r\_time

}

send\_to\_voting\_set(thread\_id, msg)

def send\_to\_voting\_set(thread\_id, msg):

try:

gen = voting\_set(thread\_id)

while 1:

send\_message(gen.next(), msg)

except StopIteration:

pass

def voting\_set(me):

for x in range(1, N + 1):

if x % n == me % n and x != me:

yield x

me = me - 1

for x in range((me / n) \* n + 1, (me / n) \* n + n + 1):

yield x

if \_\_name\_\_ == "\_\_main\_\_":

sys.exit(main())

**Show process/event call steps for algorithm that you will implement**

**Algorithm:**

* **To enter Critical section:**
  + When a site Si wants to enter the critical section, it sends a request message **REQUEST(i)** to all other sites in the request set **Ri**.
  + When a site Sj receives the request message **REQUEST(i)** from site Si, it returns a **REPLY** message to site Si if it has not sent a **REPLY** message to the site from the time it received the last **RELEASE** message. Otherwise, it queues up the request.

.

* **To execute the critical section:**
  + A site Si can enter the critical section if it has received the **REPLY** message from all the site in request set **Ri**
* **To release the critical section:**
  + When a site Si exits the critical section, it sends **RELEASE(i)** message to all other sites in request set **Ri**
  + When a site Sj receives the **RELEASE(i)** message from site Si, it send **REPLY** message to the next site waiting in the queue and deletes that entry from the queue
  + In case queue is empty, site Sj update its status to show that it has not sent any **REPLY** message since the receipt of the last **RELEASE** message

**For Handling Deadlocks**

Maekawa’s algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock. A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.

Deadlock handling requires three types of messages:

**FAILED:**

A FAILED message from site Si to site Sj indicates that Si cannot grant Sj’s request because it has currently granted permission to a site with a higher priority request.

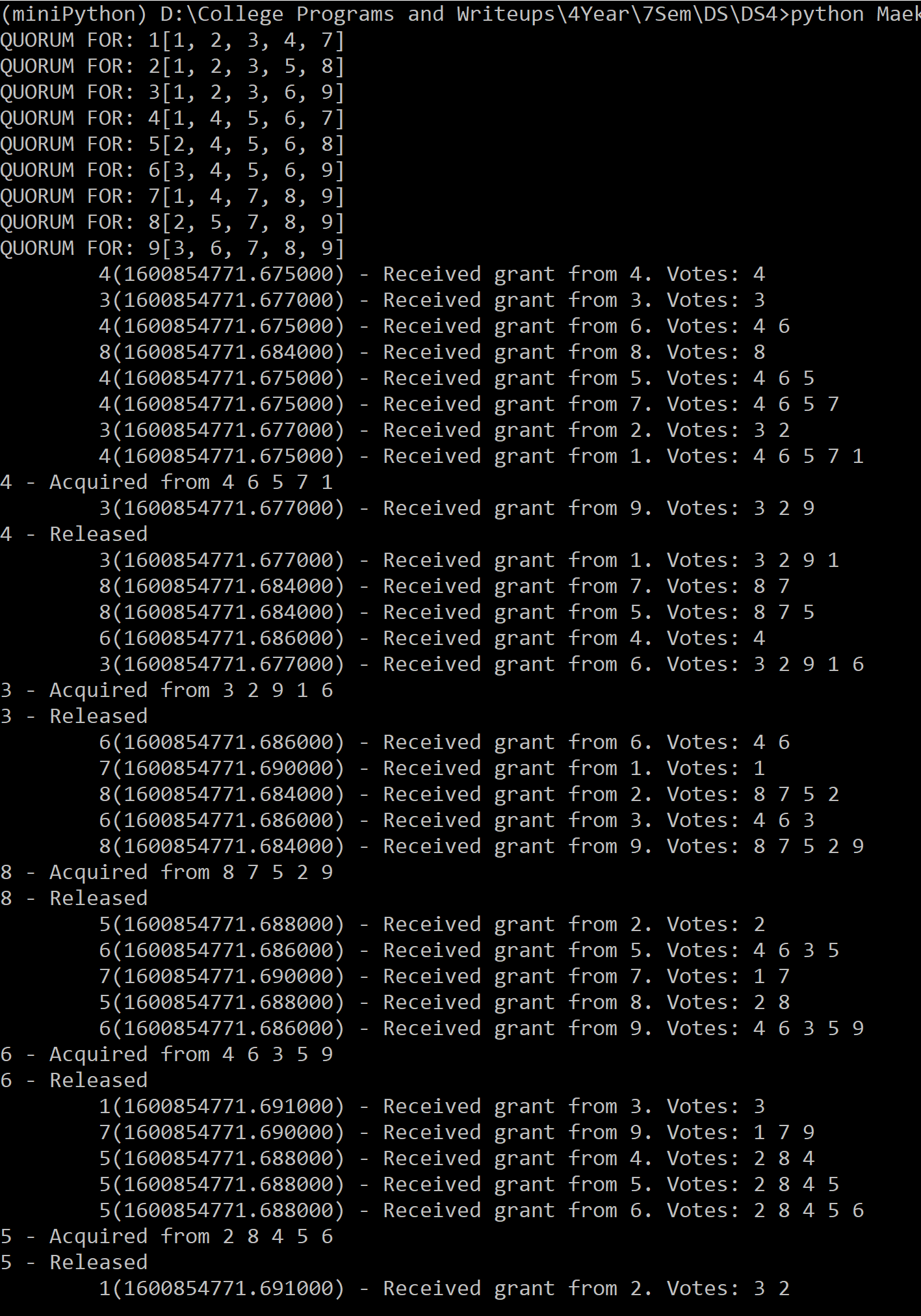
**INQUIRE:**

An INQUIRE message from Si to Sj indicates that Si would like to find out from Sj if it has succeeded in locking all the sites in its request set.

**YIELD**:

A YIELD message from site Si to Sj indicates that Si is returning the permission to Sj (to yield to a higher priority request at Sj).

**Test Procedure for objective validation**



**Conclusion:**

Studied Maekawa Algorithm for solving the critical section problem in Distributed systems.

**Evaluation**

Grading is performed for this assignment.

|  |  |
| --- | --- |
| **Election Algo laboratory work grading details Points 5 + Instructor Signature** | **Requirements** |
| 5 | * Mutual Exclusion Algorithm implement with suitable system model (1M) * Performance Analysis w.r.t. given questions: (2M) * Documentation (2M) |

|  |  |
| --- | --- |
| **Feedback** | **Yes/No** |
| * Contents in this write up has been useful to perform experiment? * Level of understanding DS has improved? |  |

**References:**

1. *A. Tanenbaum and M. Steen, Distributed systems: principles and paradigms, Prentice Hall, Second Edition, 2005, ISBN: 0132392275.*
2. *Coulouris, G., Dollimore, J., Kindberg, T., and Blair G., Distributed Systems: Concepts and Design, Addison-Wesley,* [**Fifth Edition,** 2011, ISBN: 0132143011](http://www.cdk5.net/wp/).