CS 425 / ECE 428
Distributed Systems
Fall 2015

Indranil Gupta (Indy)

Oct 1, 2015

Lecture 12: Mutual Exclusion

Central Solution

- Elect a central master (or leader)

- Master keeps
  - A **queue** of waiting requests from processes who wish to access the CS
  - A special **token** which allows its holder to access CS
- Actions of any process in group:
  - enter()
    - Send a request to master
    - Wait for token from master
  - exit()
    - Send back token to master

Central Solution

- Master Actions:
  - On receiving a request from process P*i*

    if (master has token)

    Send token to P*i*

    else

    Add P*i* to queue
  - On receiving a token from process P*i*

    if (queue is not empty)

    Dequeue head of queue (say P*j*), send that process the token

    else

    Retain token

Analysis of Central Algorithm

- Safety – at most one process in CS
  - Exactly one token
- Liveness – every request for CS granted eventually
  - With N processes in system, queue has at most N processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- FIFO Ordering is guaranteed, in order of requests received at master

Analyzing Performance

Efficient mutual exclusion algorithms use fewer messages, and make processes wait for shorter durations to access resources. Three metrics:

- ***Overhead***: the total number of messages sent in each *enter* and *exit* operation.
- ***Client delay***: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)

  (We will prefer mostly the enter operation.)
- ***Synchronization delay***: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)

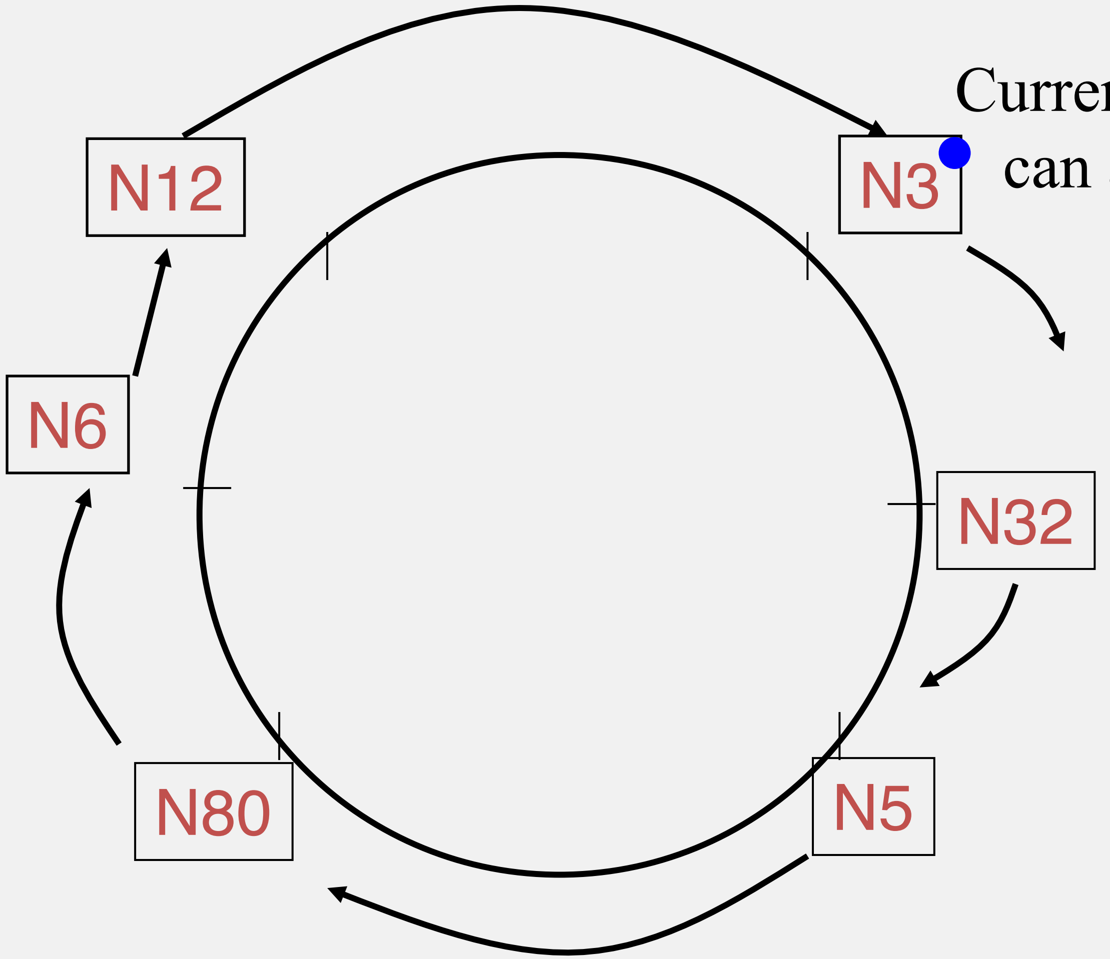
Analysis of Central Algorithm

- ***Bandwidth***: the total number of messages sent in each *enter* and *exit* operation.
  - 2 messages for enter
  - 1 message for exit
- ***Client delay***: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
  - 2 message latencies (request + grant)
- ***Synchronization delay***: the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)
  - 2 message latencies (release + grant)

But…

- The master is the performance bottleneck and SPoF (single point of failure)

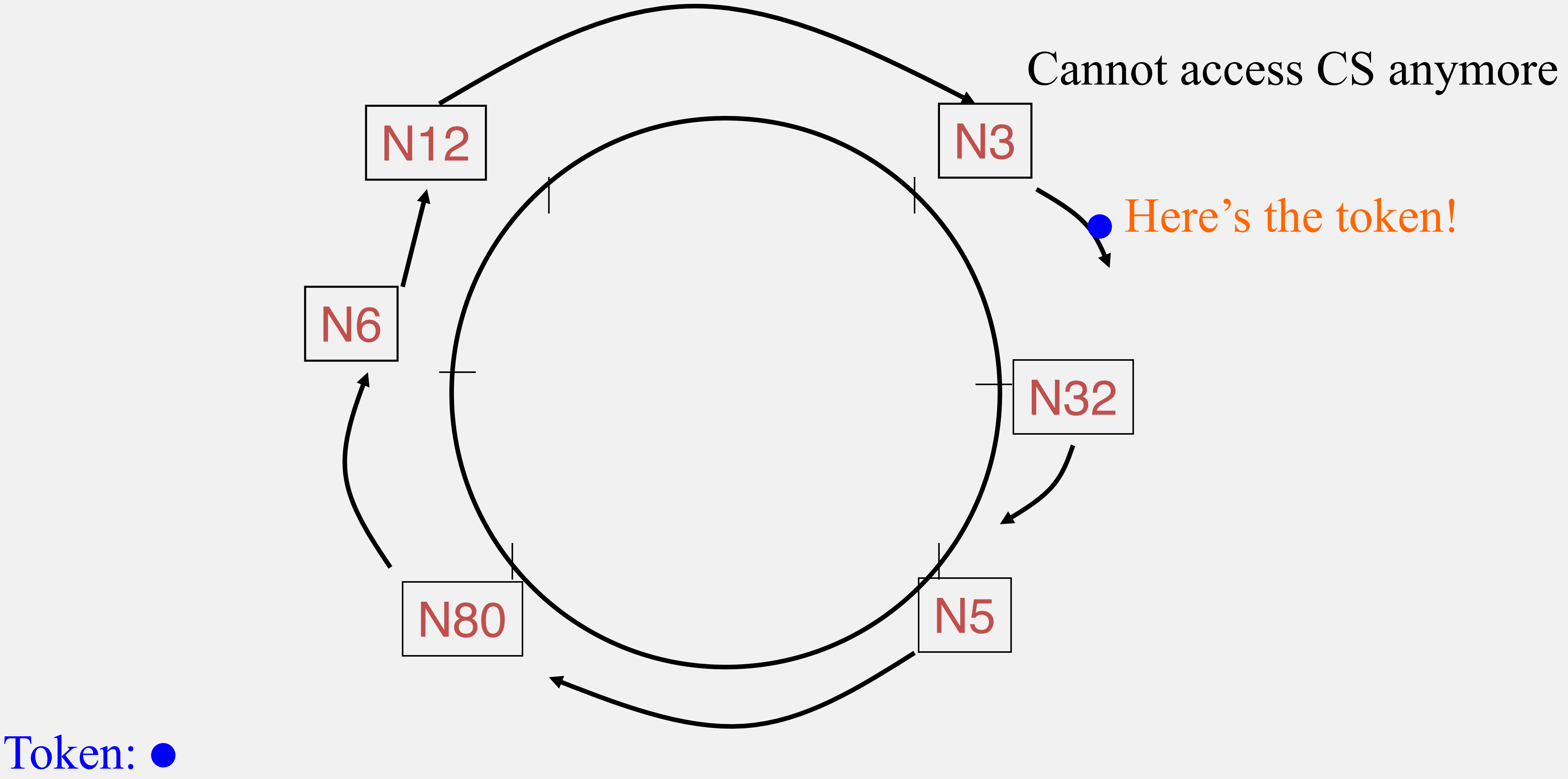
Ring-based Mutual Exclusion



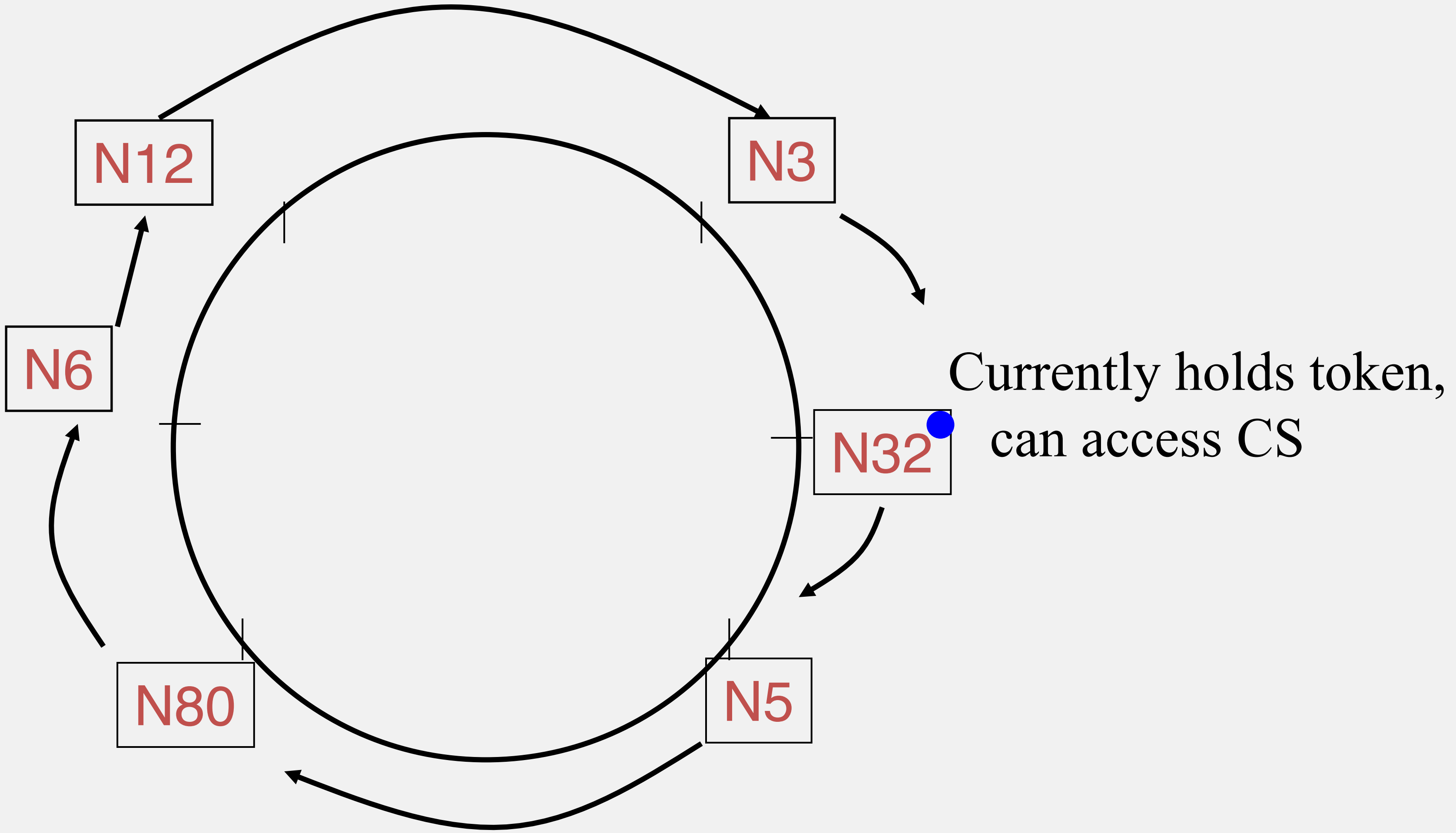
Currently holds token, can access CS

Token: ●

Ring-based Mutual Exclusion



Ring-based Mutual Exclusion



Token: ●

Ring-based Mutual Exclusion

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

Analysis of Ring-based Mutual Exclusion

- Safety
  - Exactly one token
- Liveness
  - Token eventually loops around ring and reaches requesting process (no failures)
- Bandwidth
  - Per enter(), 1 message by requesting process but up to N messages throughout system
  - 1 message sent per exit()

Analysis of Ring-Based Mutual Exclusion (2)

- Client delay: 0 to *N* message transmissions after entering enter()
    - Best case: already have token
    - Worst case: just sent token to neighbor
- Synchronization delay between one process' exit() from the CS and the next process' enter():
    - Between 1 and (*N-1*) message transmissions.
    - Best case: process in enter() is successor of process in exit()
    - Worst case: process in enter() is predecessor of process in exit()

Ricart-Agrawala’s Algorithm

- Classical algorithm from 1981
- No token

Key Idea: Ricart-Agrawala Algorithm

- enter() at process P*i*
  - multicast a request to all processes
    - Request: <T, P*i*>, where T = current Lamport timestamp at P*i*
  - Wait until *all* other processes have responded positively to request
- <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 Algorithm

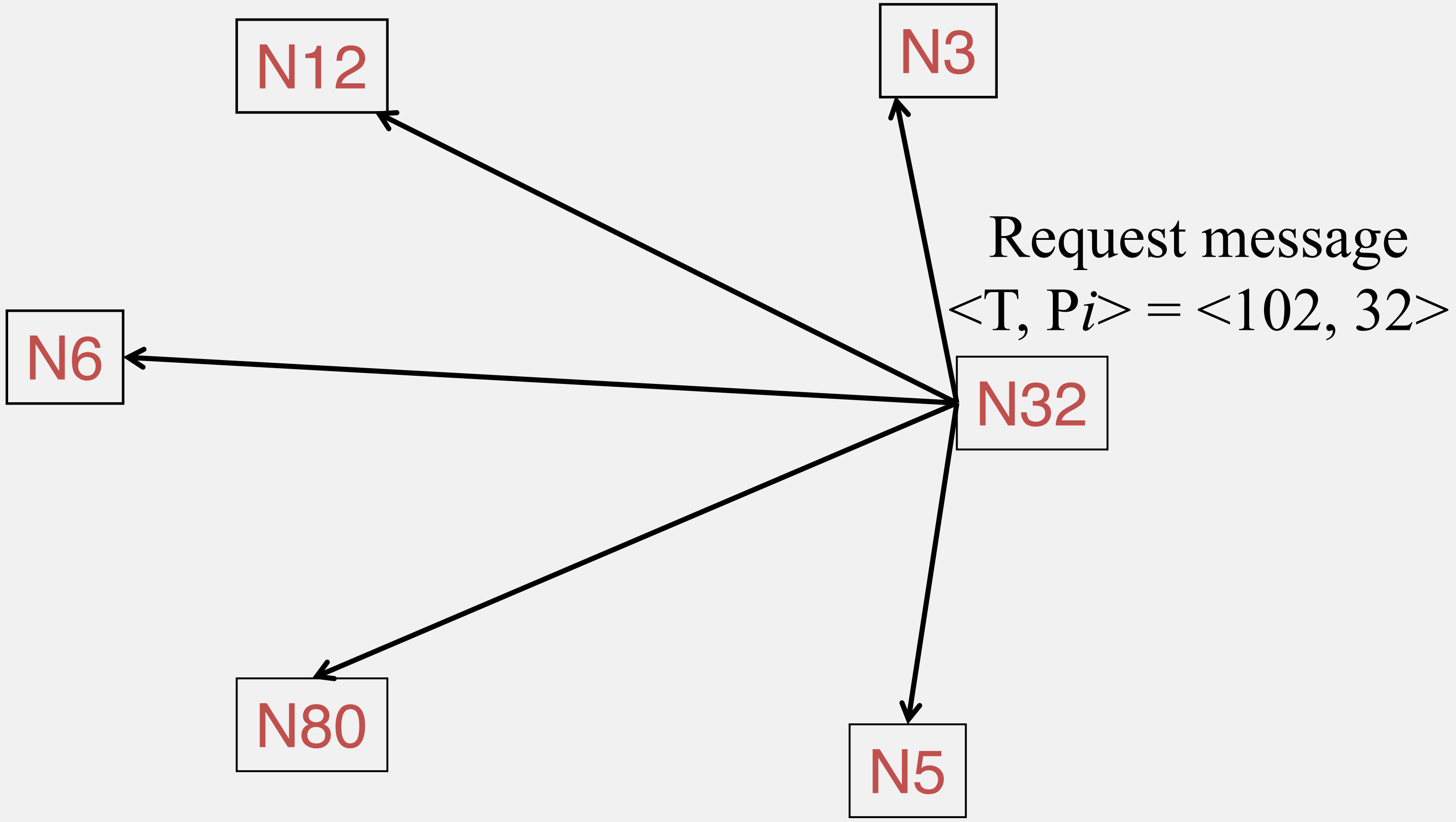
- enter() at process P*i*
  - set state to Wanted
  - multicast "Request" <T*i*, P*i*> to all processes, where T*i* = current Lamport timestamp at P*i*
  - wait until ***all*** processes send back "Reply"
  - change state to Held and enter the CS
- On receipt of a Request <T*j*, P*j*> at P*i* $(i \neq j)$:
  - **if** (state = Held) or (state = Wanted & (T*i*, *i*) < (T*j*, *j*))

    // lexicographic ordering in (T*j*, P*j*)

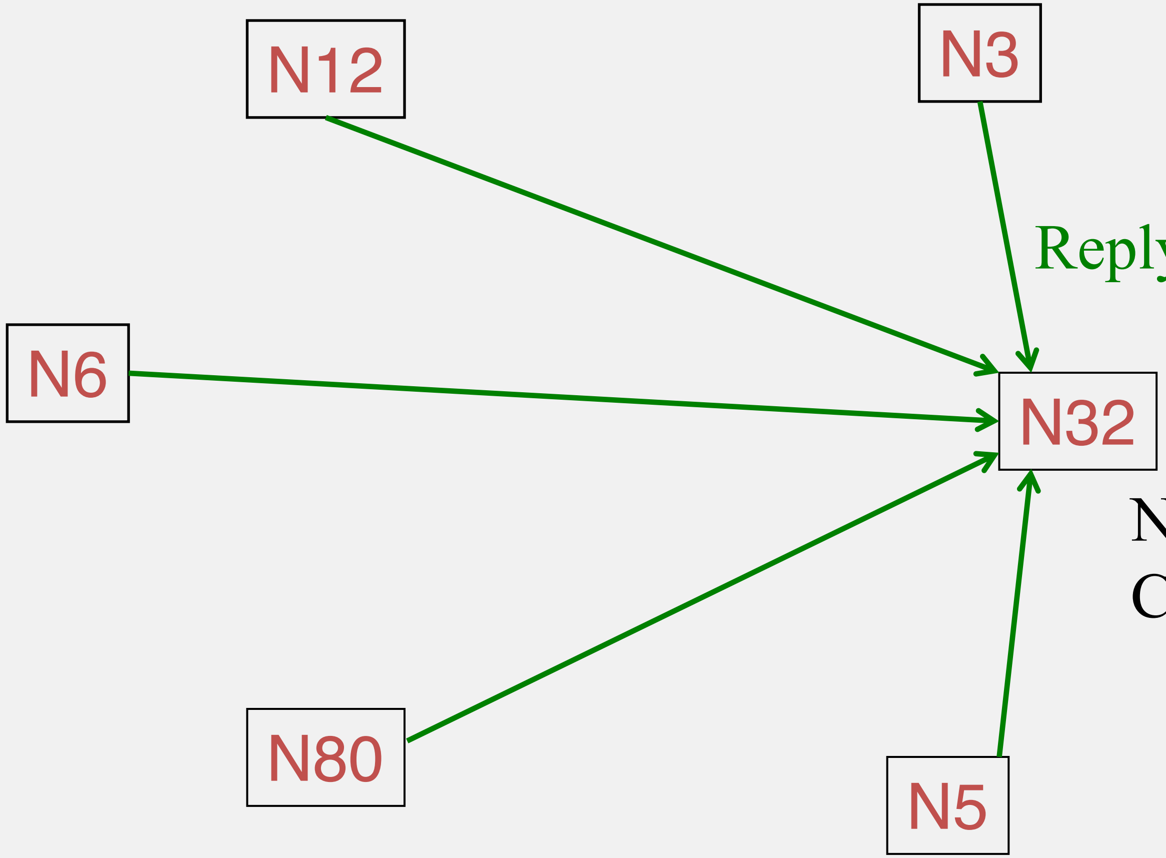
    add request to local queue (of waiting requests)

    else send "Reply" to P*j*
- exit() at process P*i*
  - change state to Released and "Reply" to *all* queued requests.

Example: Ricart-Agrawala Algorithm



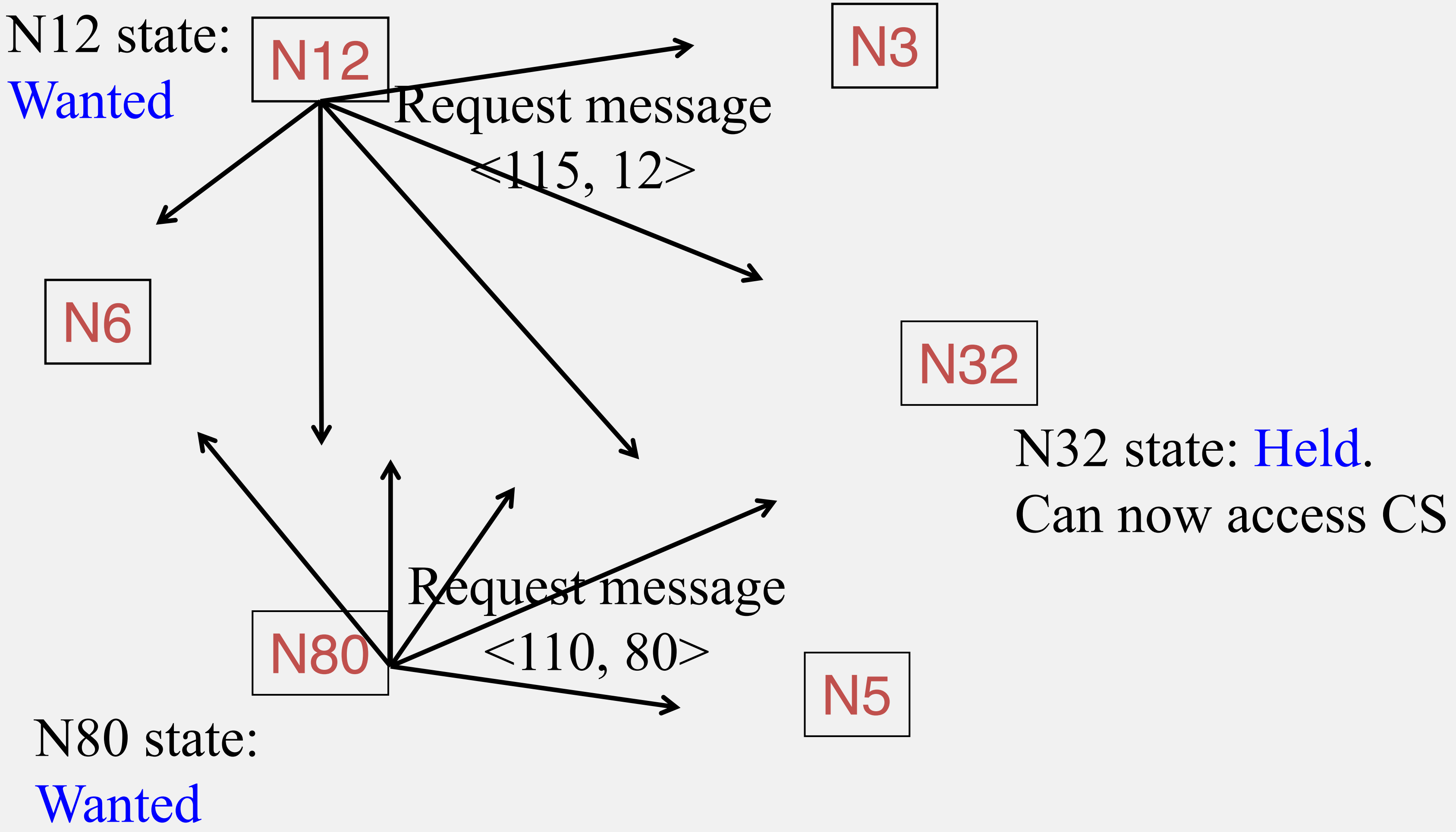
Example: Ricart-Agrawala Algorithm



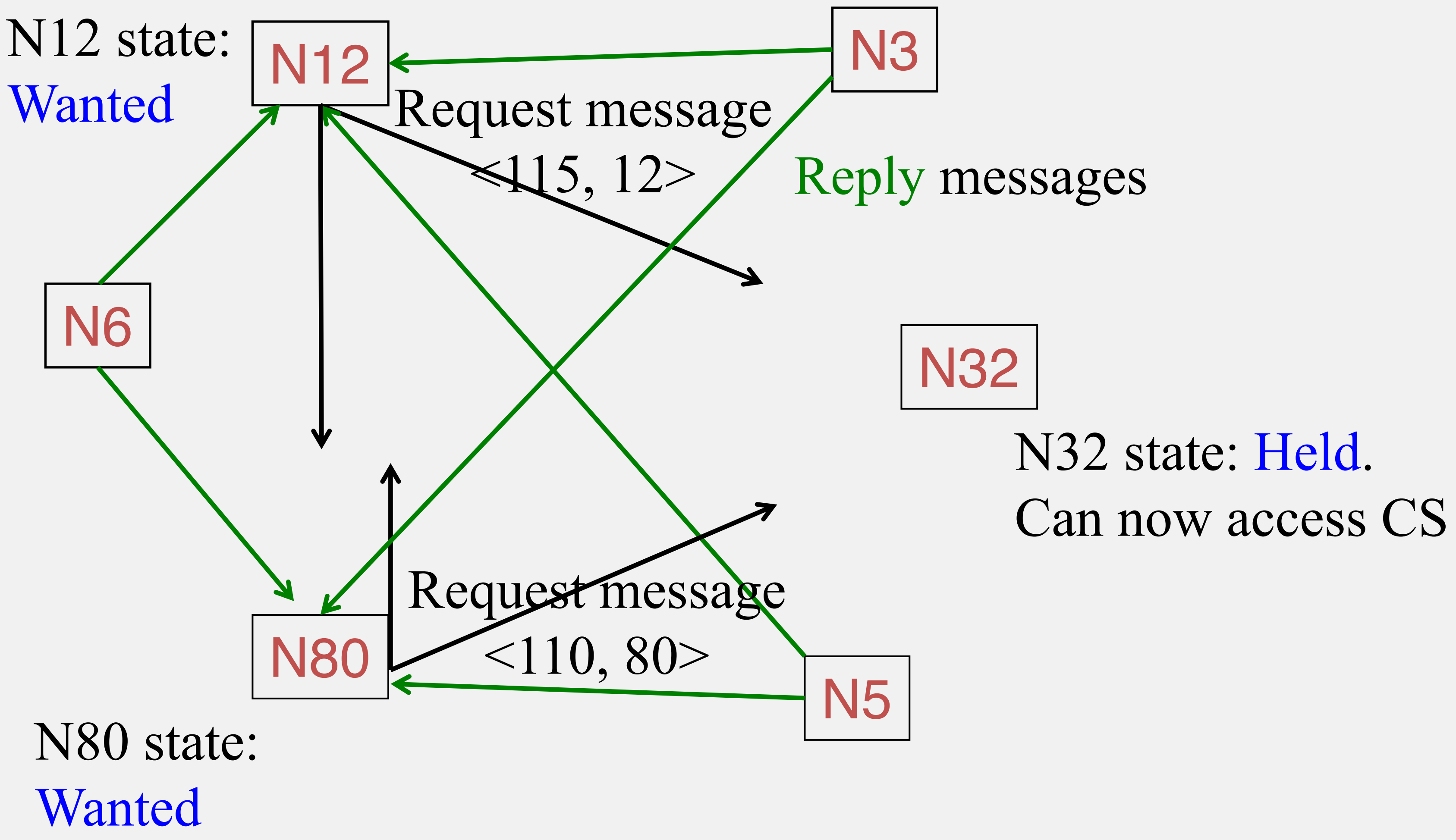
Reply messages

N32 state: Held.
Can now access CS

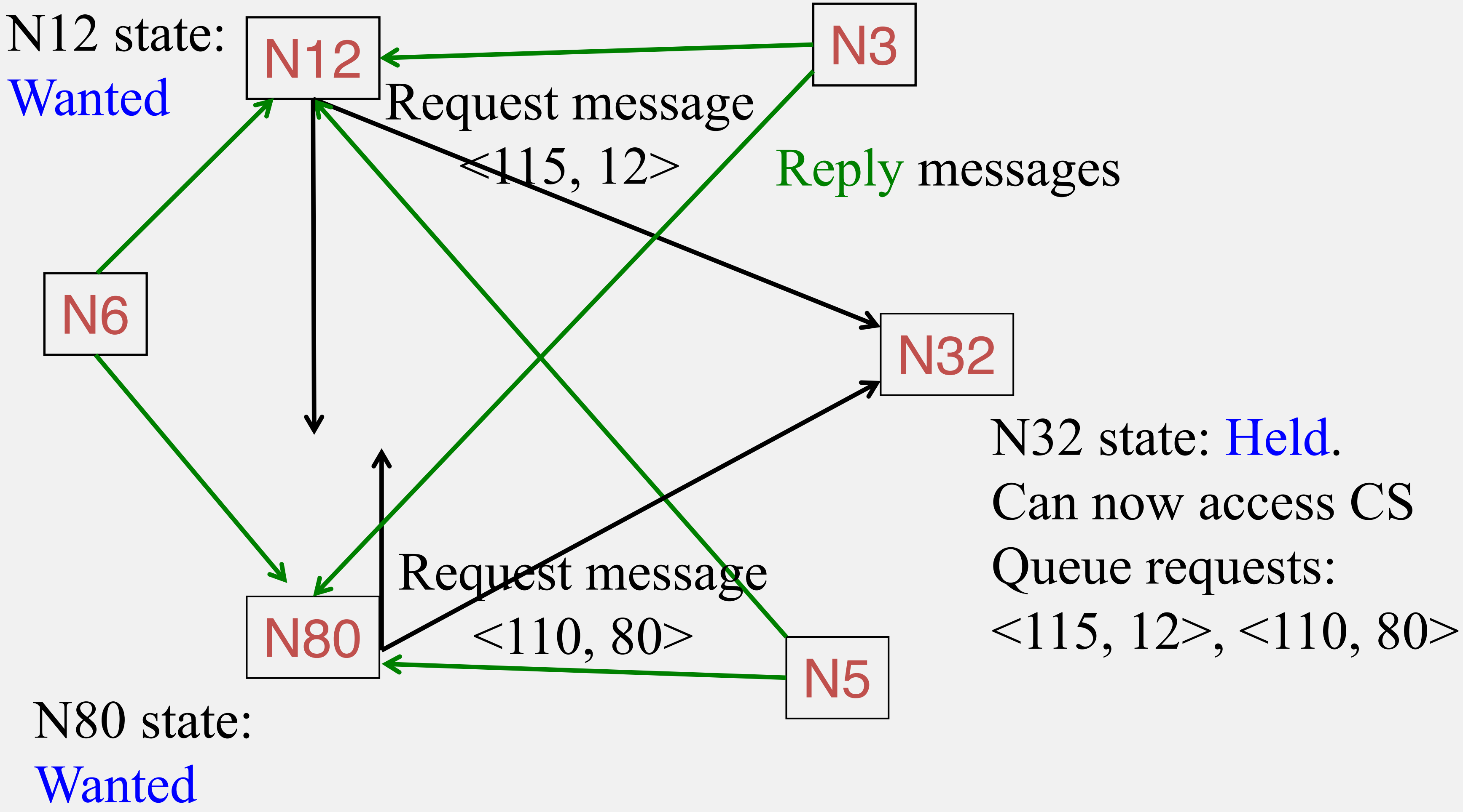
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm

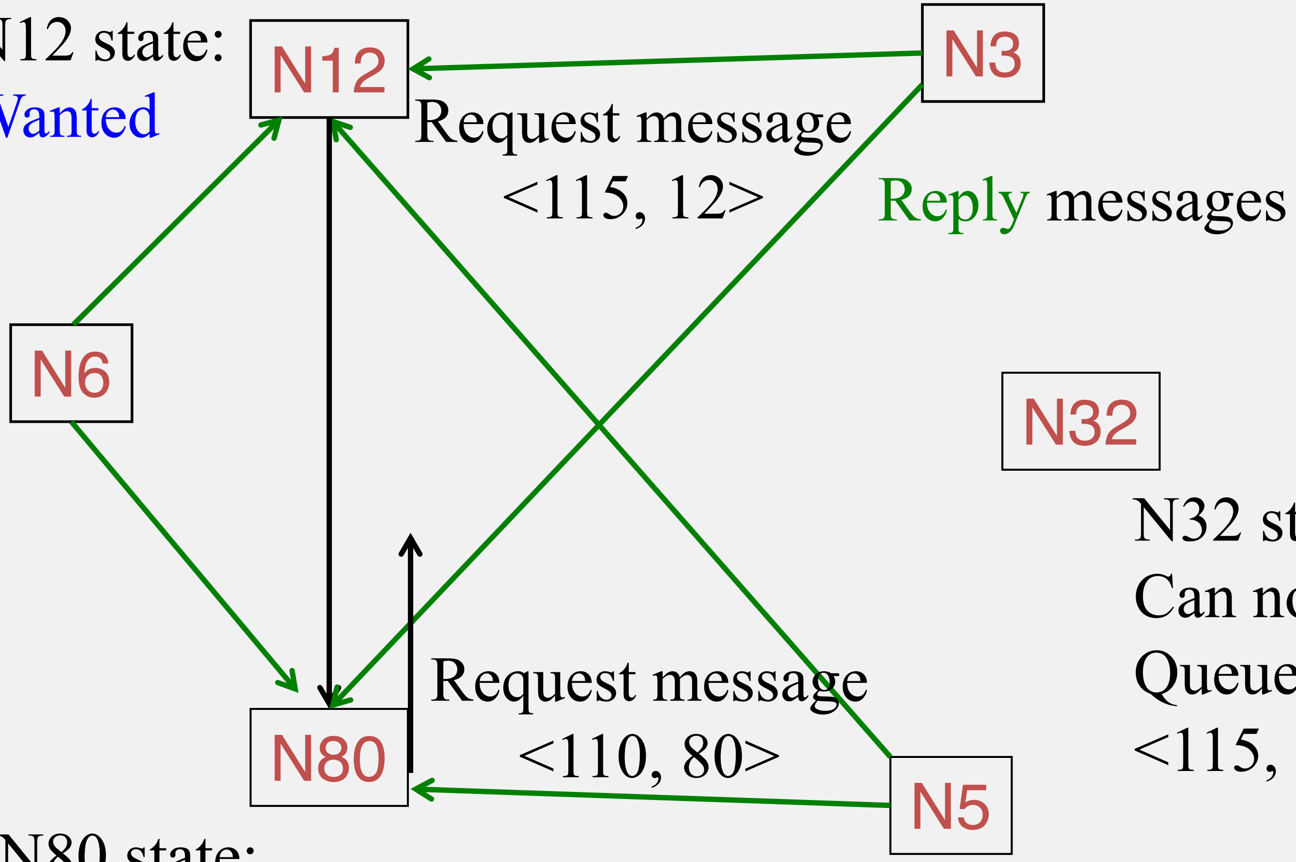


Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm

N12 state:
Wanted

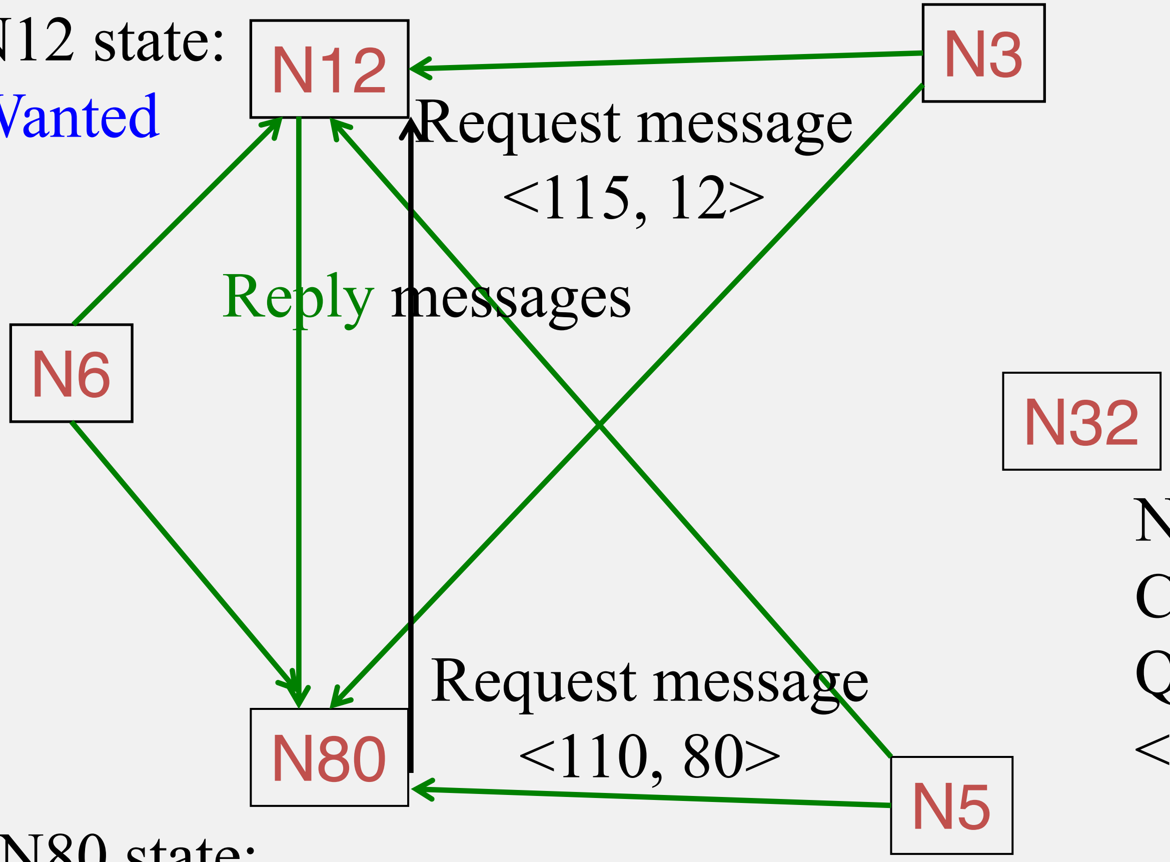


N32 state: Held.
Can now access CS
Queue requests:
<115, 12>, <110, 80>

N80 state:
Wanted
Queue requests: <115, 12> (since > (110, 80))

Example: Ricart-Agrawala Algorithm

N12 state:
Wanted

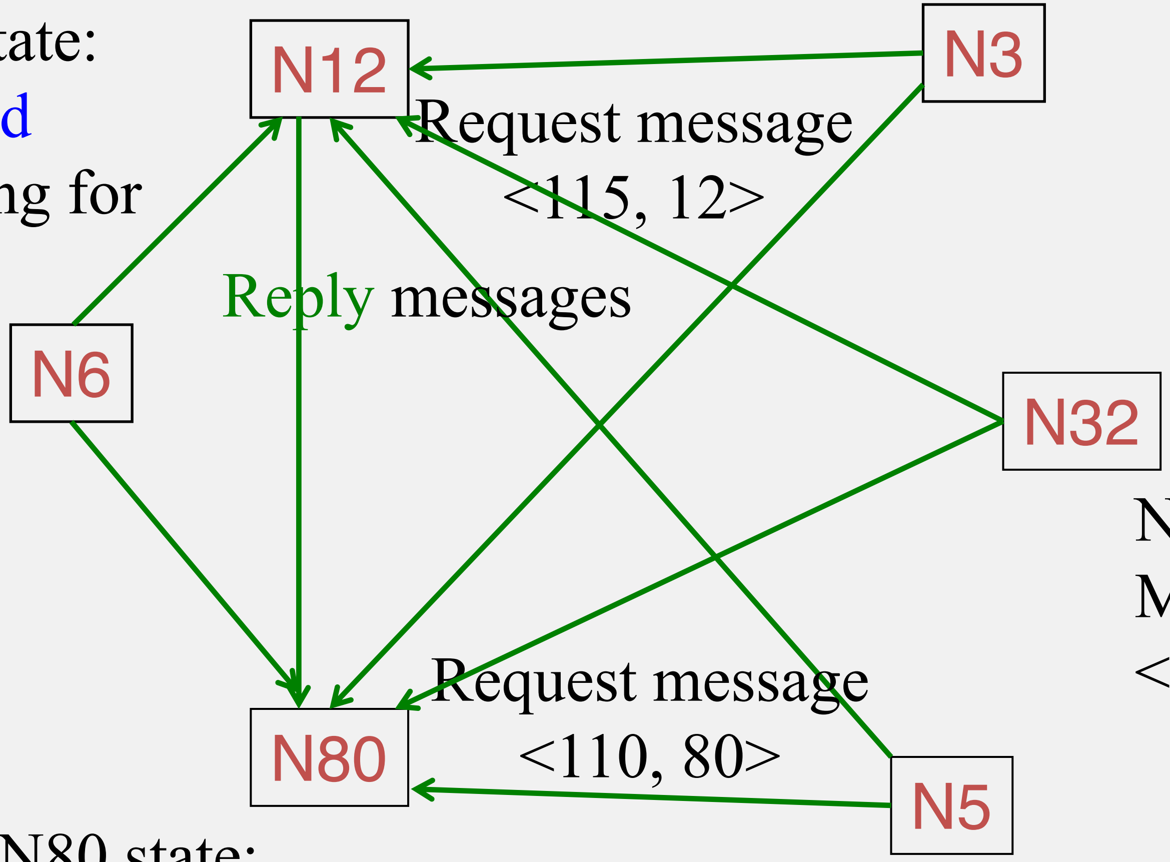


N32 state: Held.
Can now access CS
Queue requests:
<115, 12>, <110, 80>

N80 state:
Wanted
Queue requests: <115, 12>

Example: Ricart-Agrawala Algorithm

N12 state:
Wanted
(waiting for
N80's
reply)



N32 state: Released.
Multicast Reply to
<115, 12>, <110, 80>

N80 state:
Held. Can now access CS.
Queue requests: <115, 12>

Analysis: Ricart-Agrawala's Algorithm

- Safety
  - Two processes Pi and Pj cannot both have access to CS
    - If they did, then both would have sent Reply to each other
    - Thus, (Ti, i) < (Tj, j) and (Tj, j) < (Ti, i), which are together not possible
    - What if (Ti, i) < (Tj, j) and Pi replied to Pj's request before it created its own request?
      - Then it seems like both Pi and Pj would approve each others' requests
      - But then, causality and Lamport timestamps at Pi implies that Ti > Tj , which is a contradiction
      - So this situation cannot arise

Analysis: Ricart-Agrawala’s Algorithm (2)

- Liveness
  - Worst-case: wait for all other (*N-1*) processes to send Reply

Performance: Ricart-Agrawala's Algorithm

- Overhead: 2*(*N-1*) messages per enter() operation
  - *N-1* unicasts for the multicast request + *N-1* replies
  - *N* messages if the underlying network supports multicast (1 multicast + *N-1* unicast replies)
  - *N-1* unicast messages per exit operation
    - 1 multicast if the underlying network supports multicast

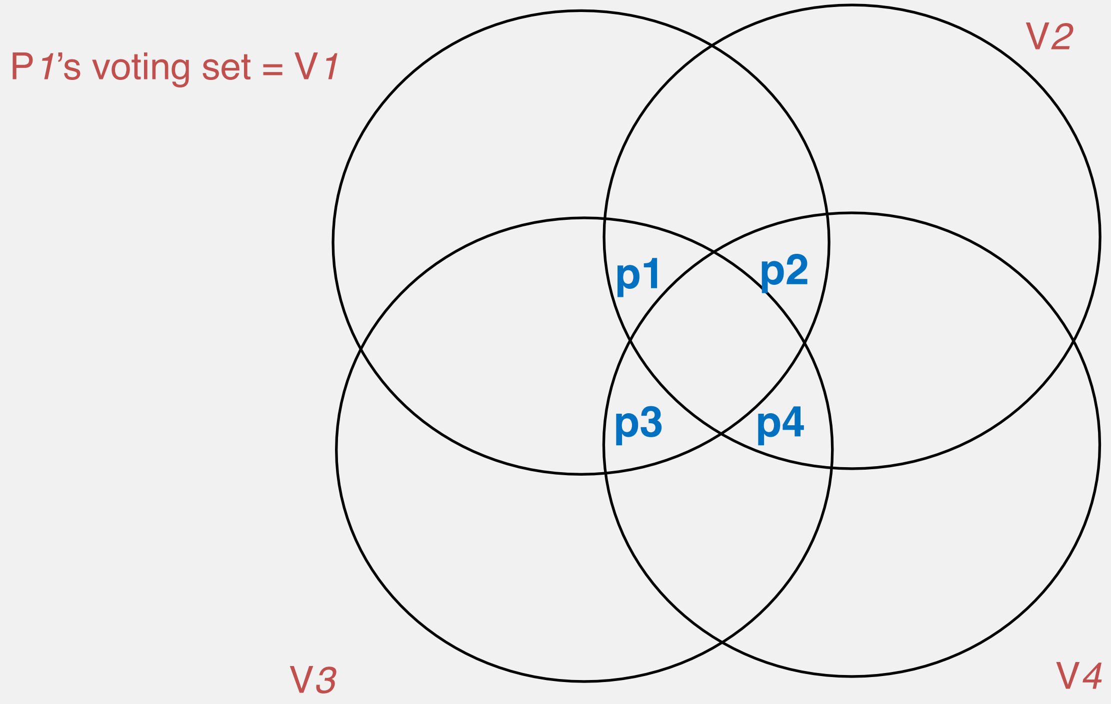
Maekawa's Algorithm: Key Idea

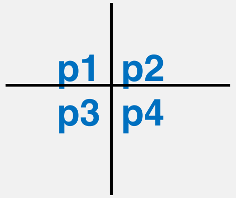
- Ricart-Agrawala requires replies from *all* processes in group
- Instead, get replies from only *some* processes in group
- But ensure that only one process is given access to CS (Critical Section) at any given time

Maekawa's Voting Sets

- Each process P*i* is associated with a <u>*voting set*</u> V*i* (of processes)
- Each process belongs to its own voting set
- *The intersection of any two voting sets must be non-empty*
  - *Same concept as Quorums*
- Each voting set is of size *K*
- Each process belongs to *M* other voting sets
- Maekawa showed that $K=M=$ *order of* $\sqrt{N}$ feasible
- One way of doing this is to put N processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix and for each P*i*, its voting set V*i* = row containing P*i* + column containing P*i*. Size of voting set = $2*\sqrt{N}-1$

Example: Voting Sets with N=4





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*

Actions (2)

- 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*:

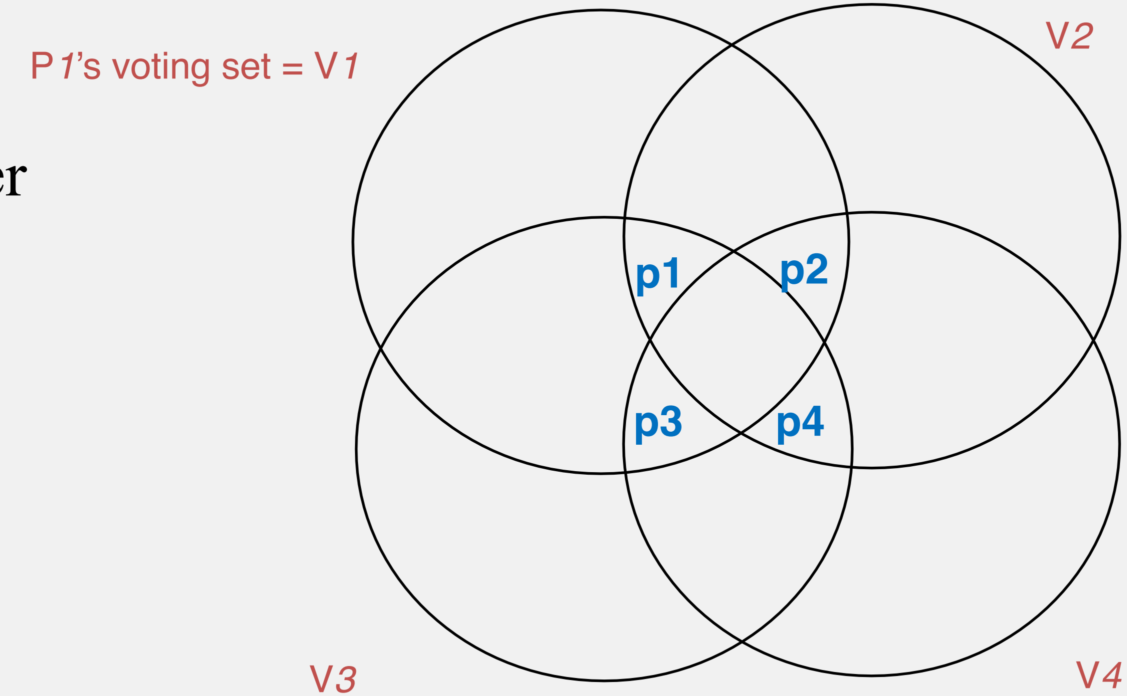
if (queue empty)
    voted = false
else
    dequeue head of queue, say P*k*
    Send Reply *only* to P*k*
    voted = true

Safety

- When a process P*i* receives replies from all its voting set V*i* members, no other process P*j* could have received replies from all its voting set members V*j*
    - V*i* and V*j* intersect in at least one process say P*k*
    - But P*k* sends only one Reply (vote) at a time, so it could not have voted for both P*i* and P*j*

Liveness

- A process needs to wait for at most (*N-1*) other processes to finish CS
- But does not guarantee liveness
- Since can have a *deadlock*
- Example: all 4 processes need access
  - P1 is waiting for P3
  - P3 is waiting for P4
  - P4 is waiting for P2
  - P2 is waiting for P1
  - No progress in the system!
- There are deadlock-free versions



Performance

- Overhead
  - $2\sqrt{N}$ messages per enter()
  - $\sqrt{N}$ messages per exit()
  - Better than Ricart and Agrawala's ($2*(N-1)$ and $N-1$ messages)
  - $\sqrt{N}$ quite small. $N \sim 1$ million => $\sqrt{N} = 1K$

Summary

- Mutual exclusion important problem in cloud computing systems
- Classical algorithms
  - Central
  - Ring-based
  - Ricart-Agrawala
  - Maekawa