Distributed Mutual Exclusion

CMPS 4760/6760: Distributed Systems

Acknowledgement: slides adapted from Indranil Gupta's lecture notes:
https://courses.engr.illinois.edu/cs425/fa2019/index.html

Why Mutual Exclusion?

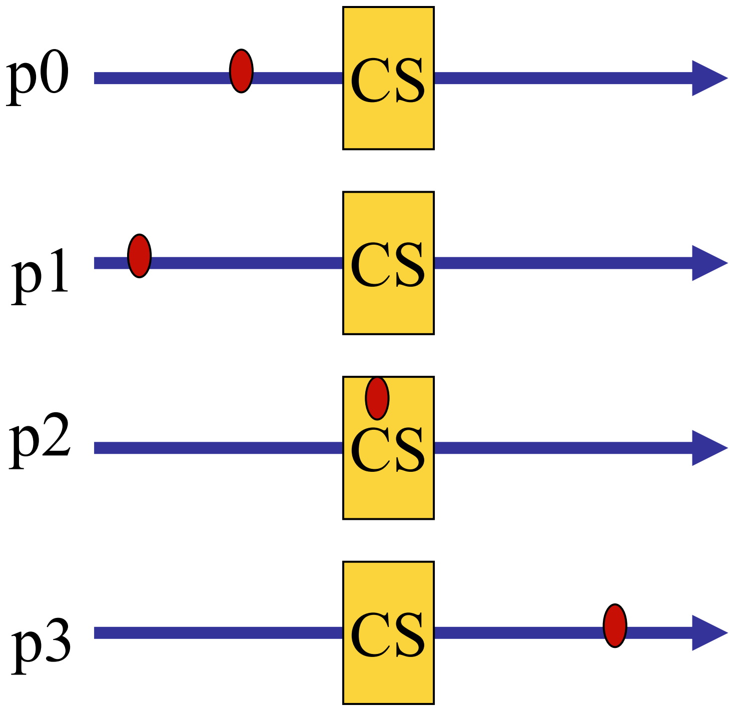
- Bank's Servers in the Cloud: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of $1000 concurrently from the bank's cloud server
  - Both ATMs add $10,000 to this amount (locally at the ATM)
  - Both write the final amount to the server
  - What's wrong?

Why Mutual Exclusion?

- Bank's Servers in the Cloud: Two of your customers make simultaneous deposits of $10,000 into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of $1000 concurrently from the bank's cloud server
  - Both ATMs add $10,000 to this amount (locally at the ATM)
  - Both write the final amount to the server
  - You lose $10,000
- The ATMs need *mutually exclusive* access to your account entry at the server
  - or, mutually exclusive access to executing the code that modifies the account entry

Problem Statement for Mutual Exclusion

- 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



Approaches to Solve Mutual Exclusion

- Single OS:
  - If all processes are running in one OS on a machine (or VM), then
  - Semaphores, locks, condition variables, monitors, etc.

- Distributed system:
  - Message passing only

Problem Specification

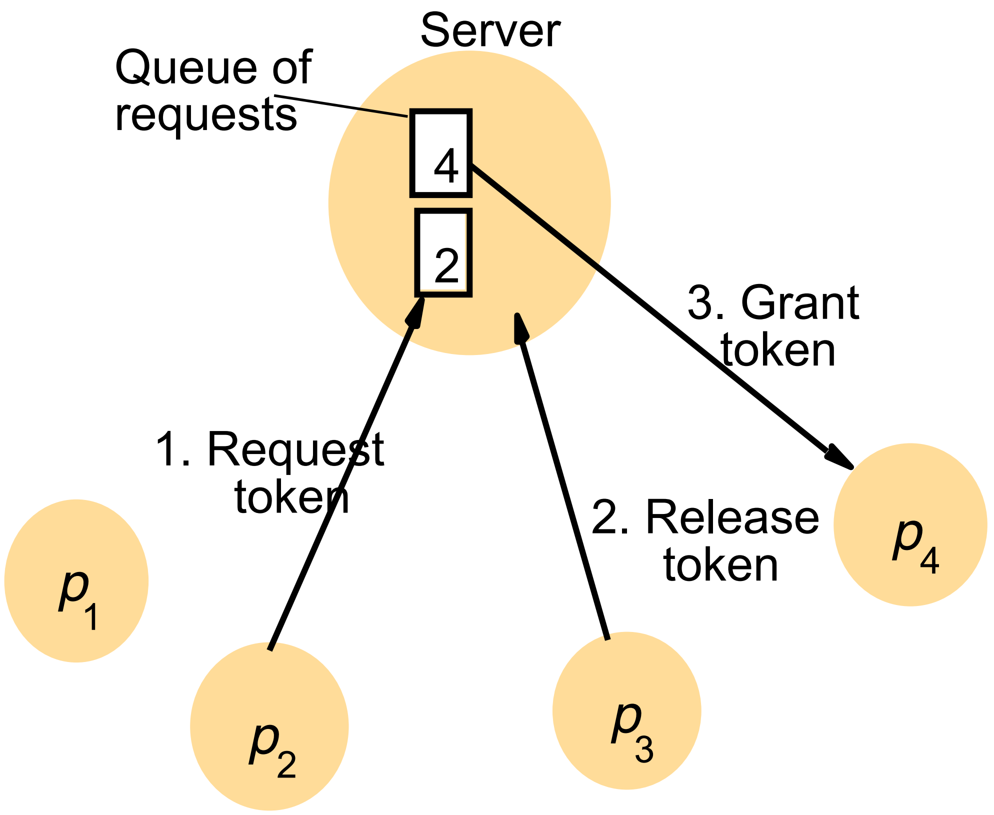
- Safety: At most one process can execute in the critical section (CS) at a time
  - Safety – nothing "bad" will happen
- Liveness: Every request for the critical section is eventually granted
  - Liveness – something "good" will eventually happen
- Fairness: Different requests are granted in the order they are made
  - If one request to enter the CS happened-before another, then entry to the CS is granted in that order

Assumptions

- No faults in the system: both processes and communication links are reliable
- A process that is granted access to the critical section eventually releases it (cooperation)
- A single critical section (CS)

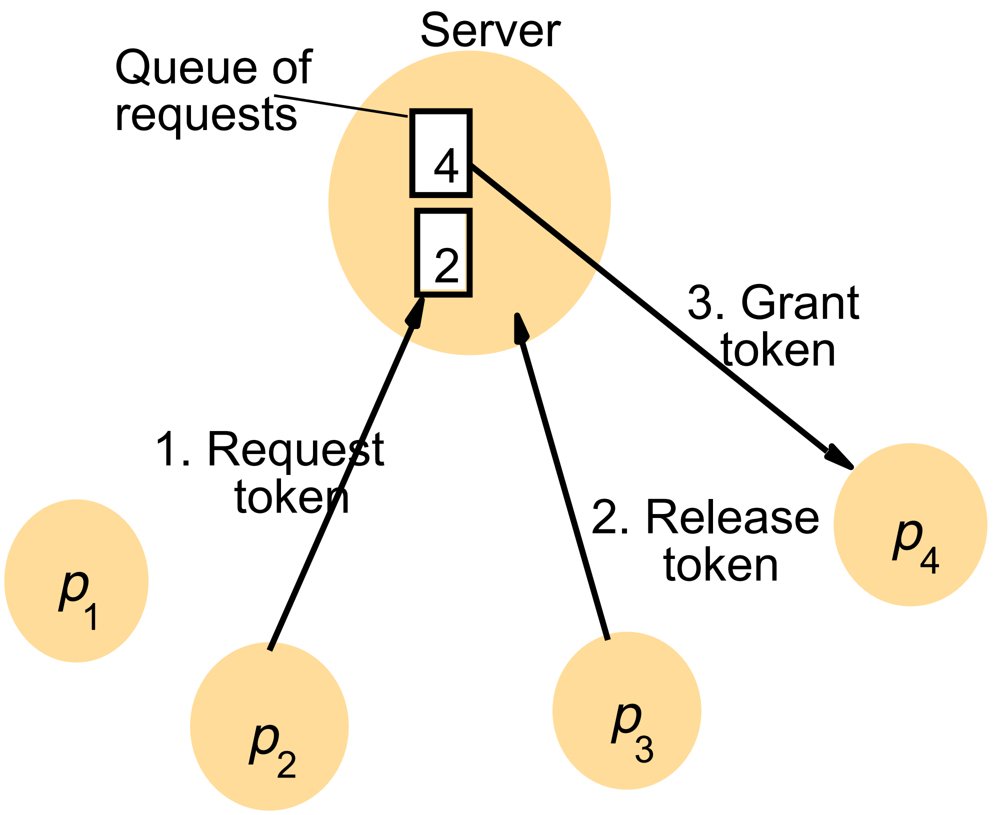
A simple centralized solution

- A server serves as the coordinator for the CS
- Any process that needs to access the CS sends a request to the coordinator
- The coordinator puts requests in a queue in the order it receives them and grants permission to the process that is at the head of the queue
- When a process exits the CS, it sends a release message to the coordinator



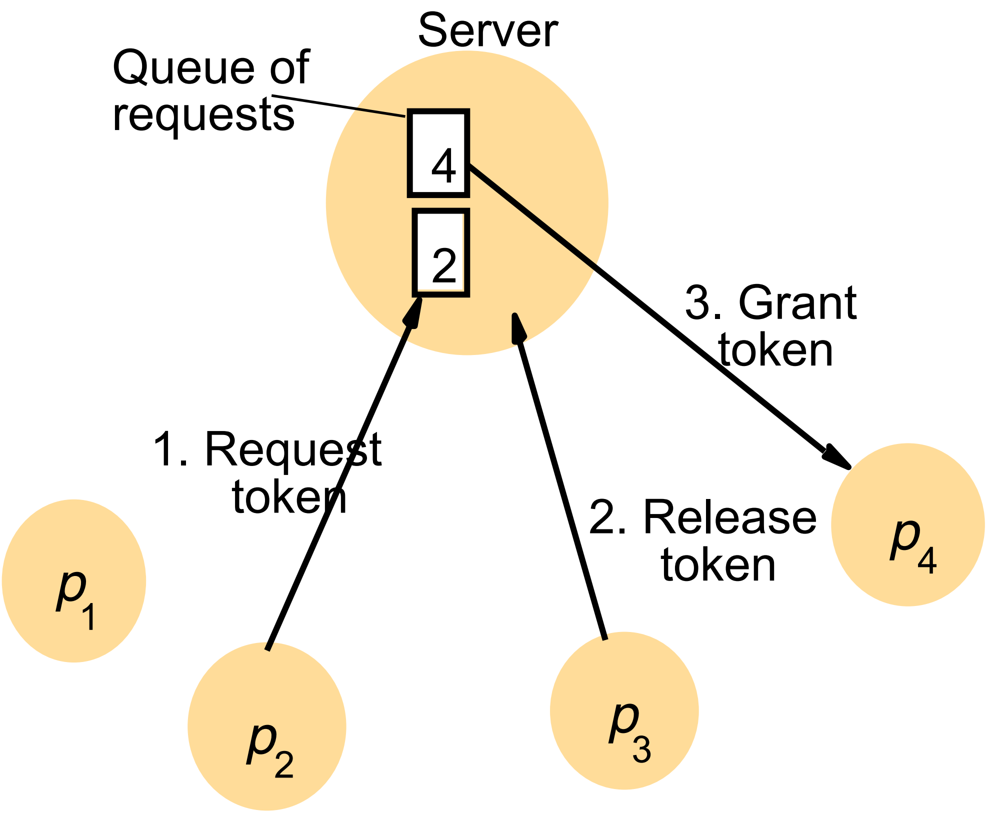
A simple centralized solution

- Assuming no faults, safety and liveness satisfied, but not fairness (why?)



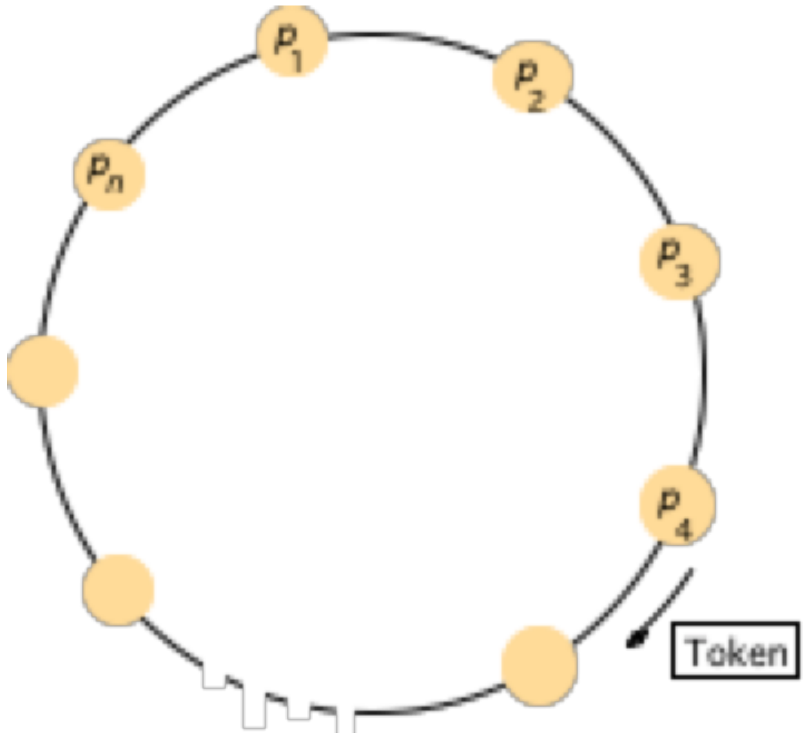
A simple centralized solution

- Performance
  - Entering the CS takes 2 messages
  - Exiting the CS takes 1 message
  - Delay to enter the CS (when *no* other process is in, or waiting)
    - 2 message latencies (request + grant) : one round-trip time
  - Synchronization delay: time interval between one process exiting the CS and the other process entering it
    - 2 message latencies (release + grant): one round-trip time



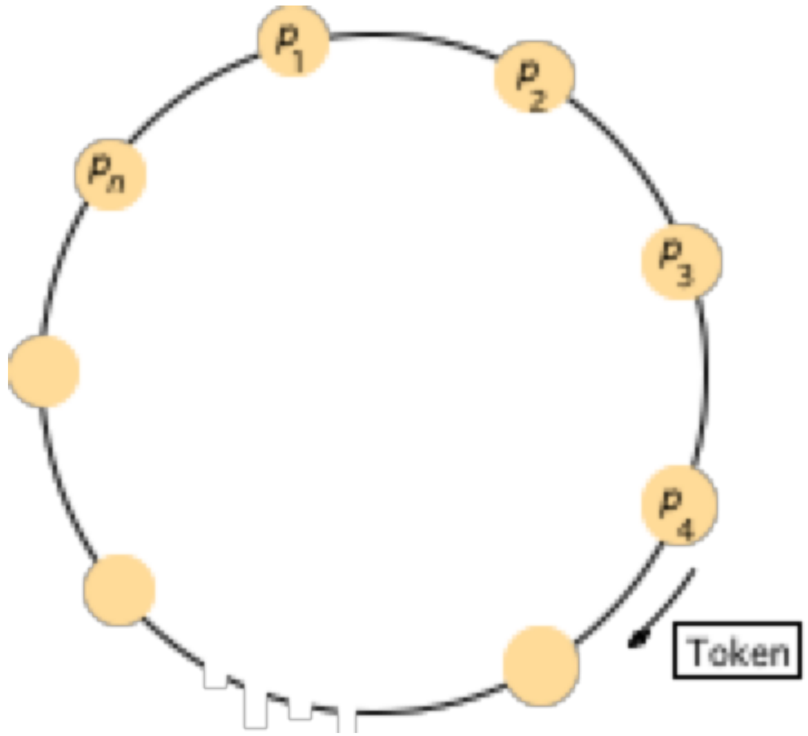
A ring-based algorithm

- Arrange processes into a logical ring
- A token passes through the processes in a single direction
- A process can access the CS when it receives the token. It forwards the token to its neighbor when it exits the CS
- If a process receives the token and does not need to access the CS, it immediately forwards the token to its neighbor



A ring-based algorithm

- Assuming no faults, safety and liveness satisfied, but not fairness (why?)
- Performance
  - Processes send and receive messages around the ring even when no one requires entry to the CS
  - Delay to enter the CS: 0 $\sim N$ messages
  - Synchronization delay: 1 $\sim N$ messages



Ricart-Agrawala Algorithm

- 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

Assumptions

- No faults in the system: both processes and communication links are reliable
- A process that is granted access to the critical section eventually releases it (cooperation)
- A single critical section (CS)
- A completely connected graph, so that every process can directly communicate with every other process in the system

Key Idea: Ricart-Agrawala Algorithm

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

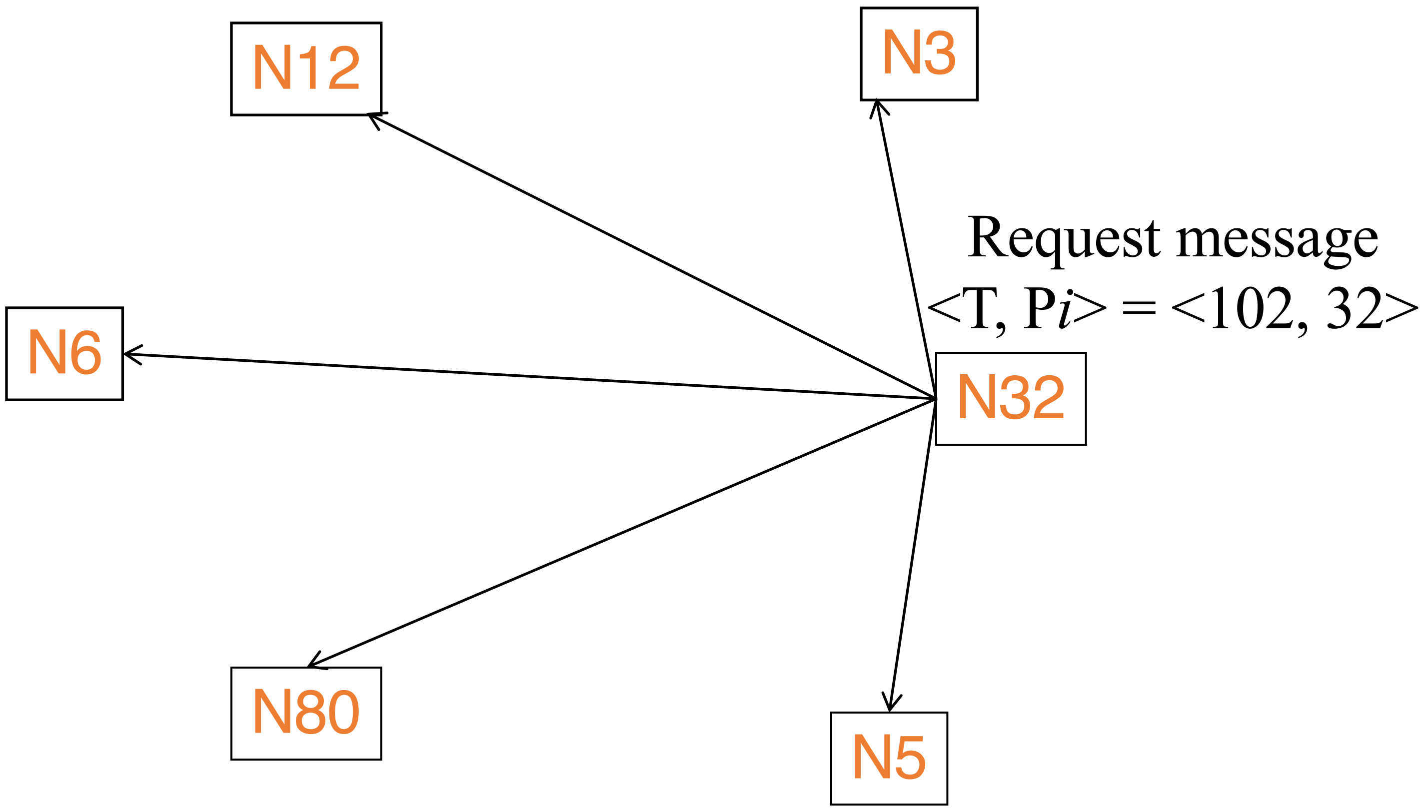
Messages in RA Algorithm

- 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)

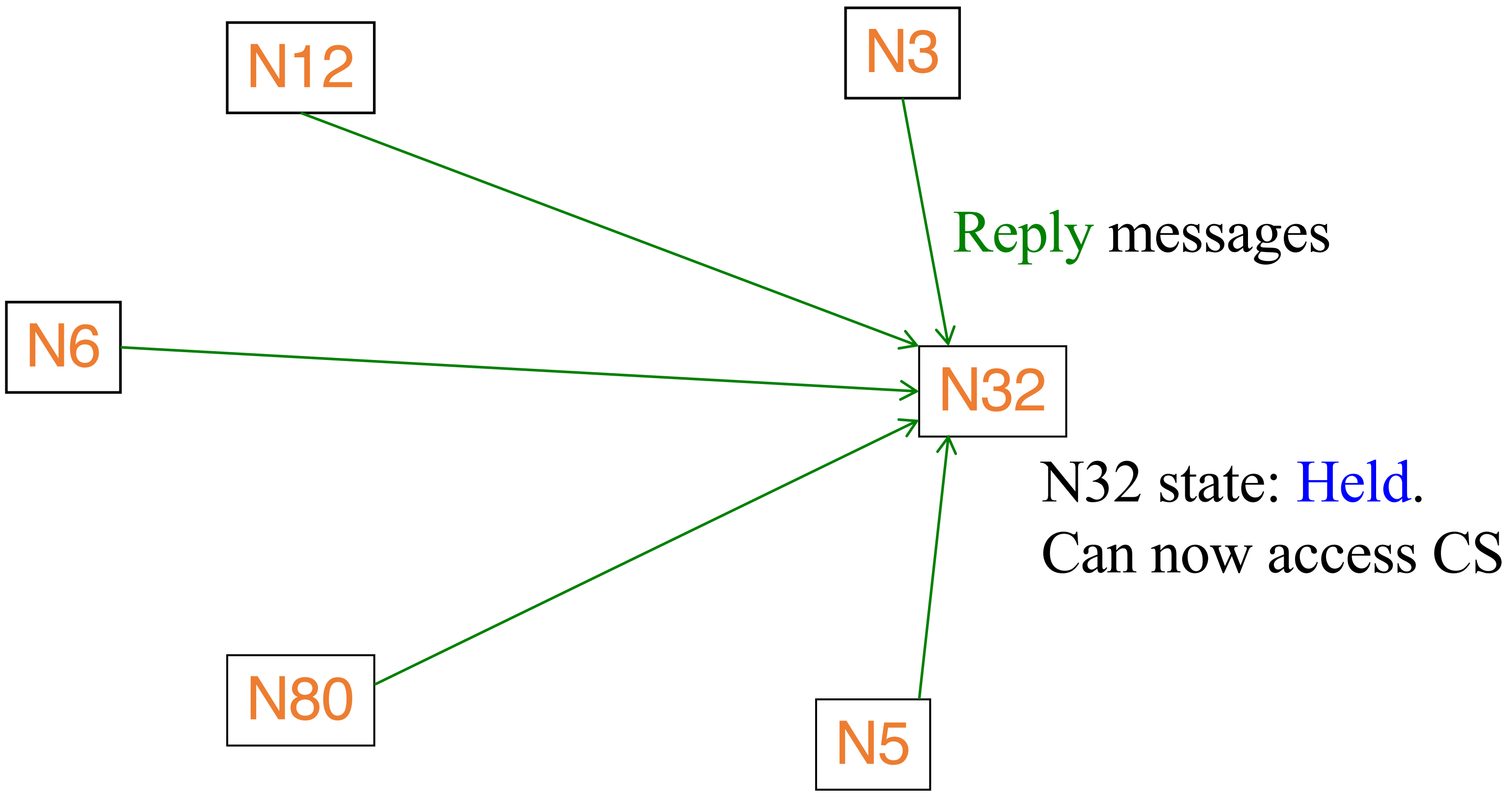
    add request to local queue (of waiting requests)

    else send "Reply" to P_j
- exit() at process P_i
  - change state to Released and "Reply" to all queued requests

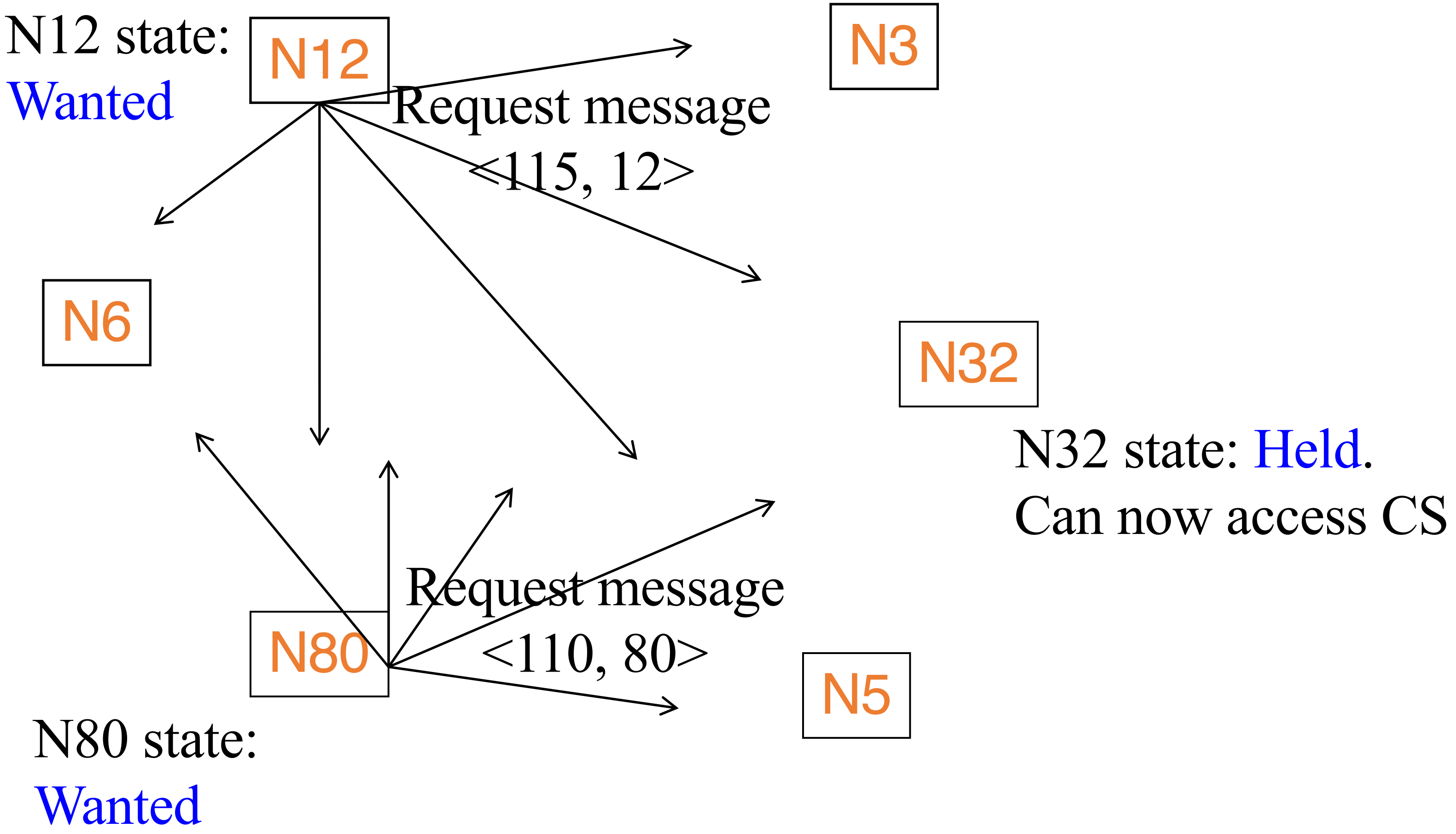
Example: Ricart-Agrawala Algorithm



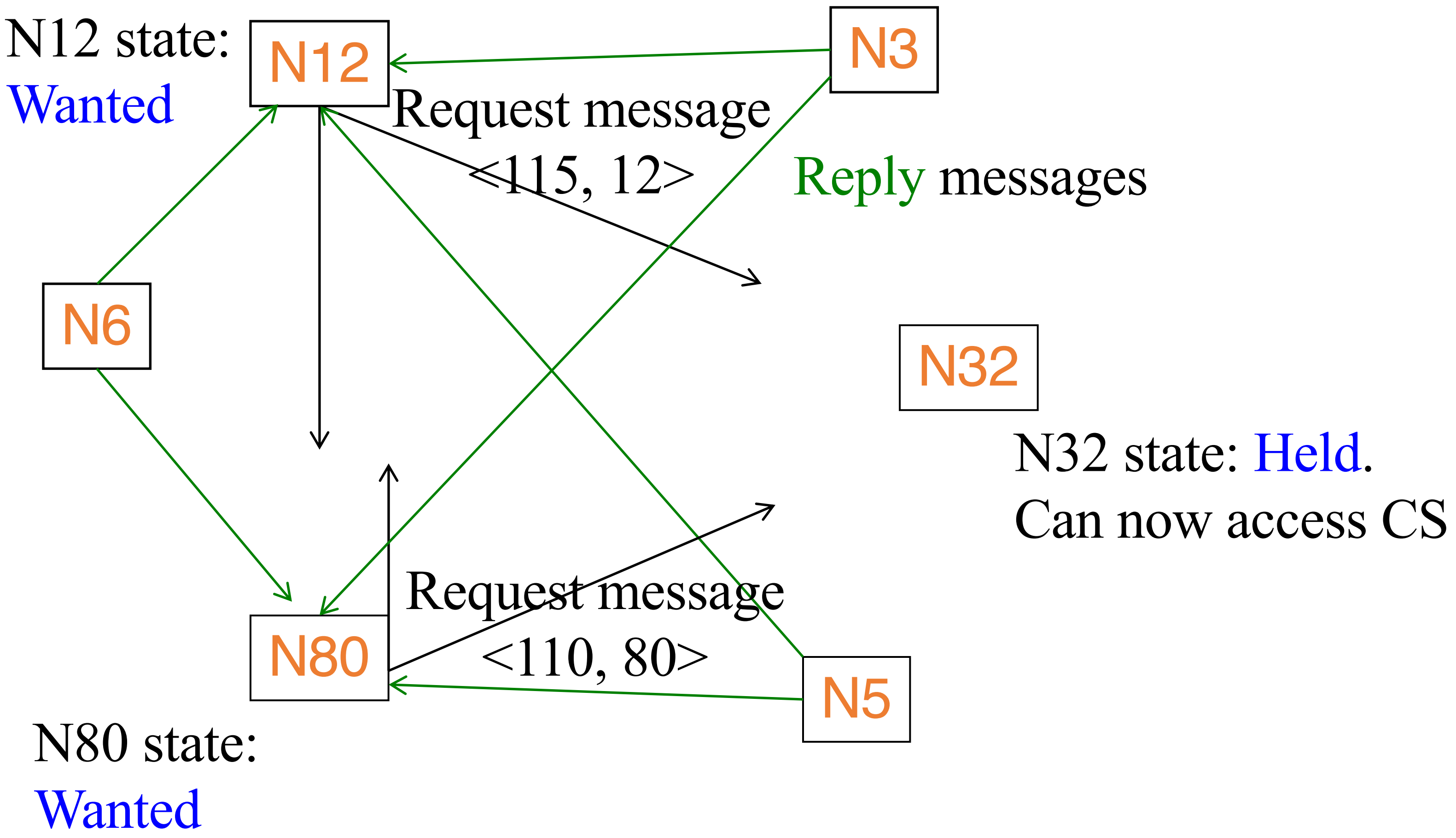
Example: Ricart-Agrawala Algorithm



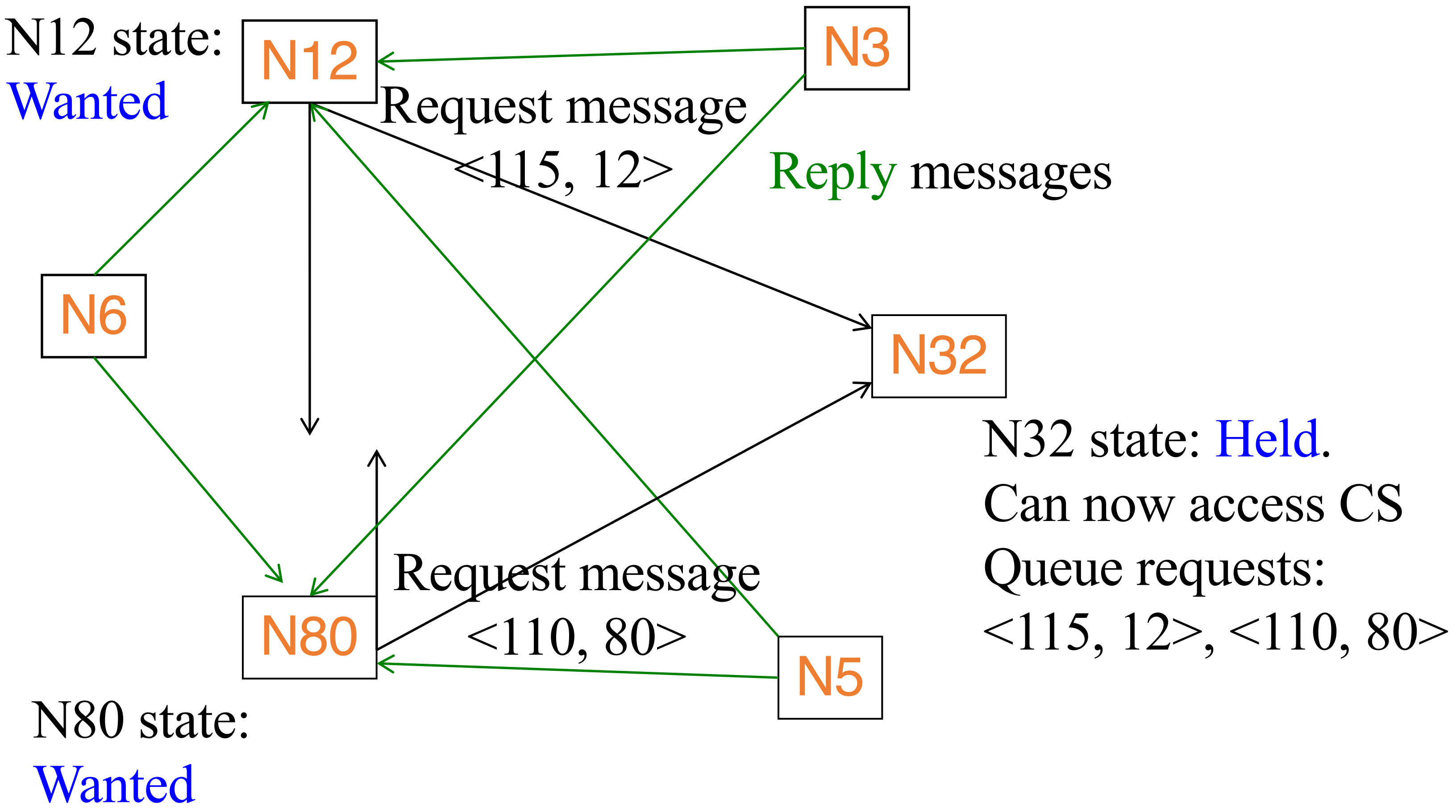
Example: Ricart-Agrawala Algorithm



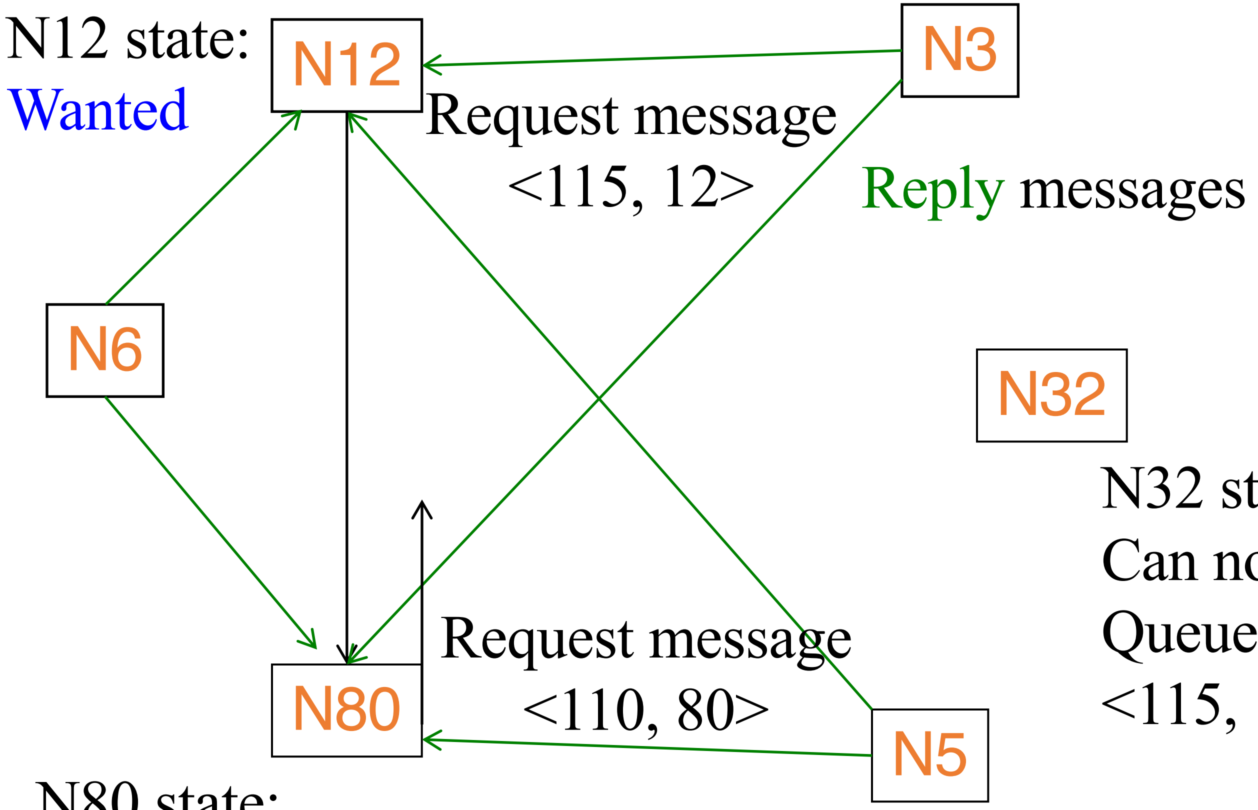
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



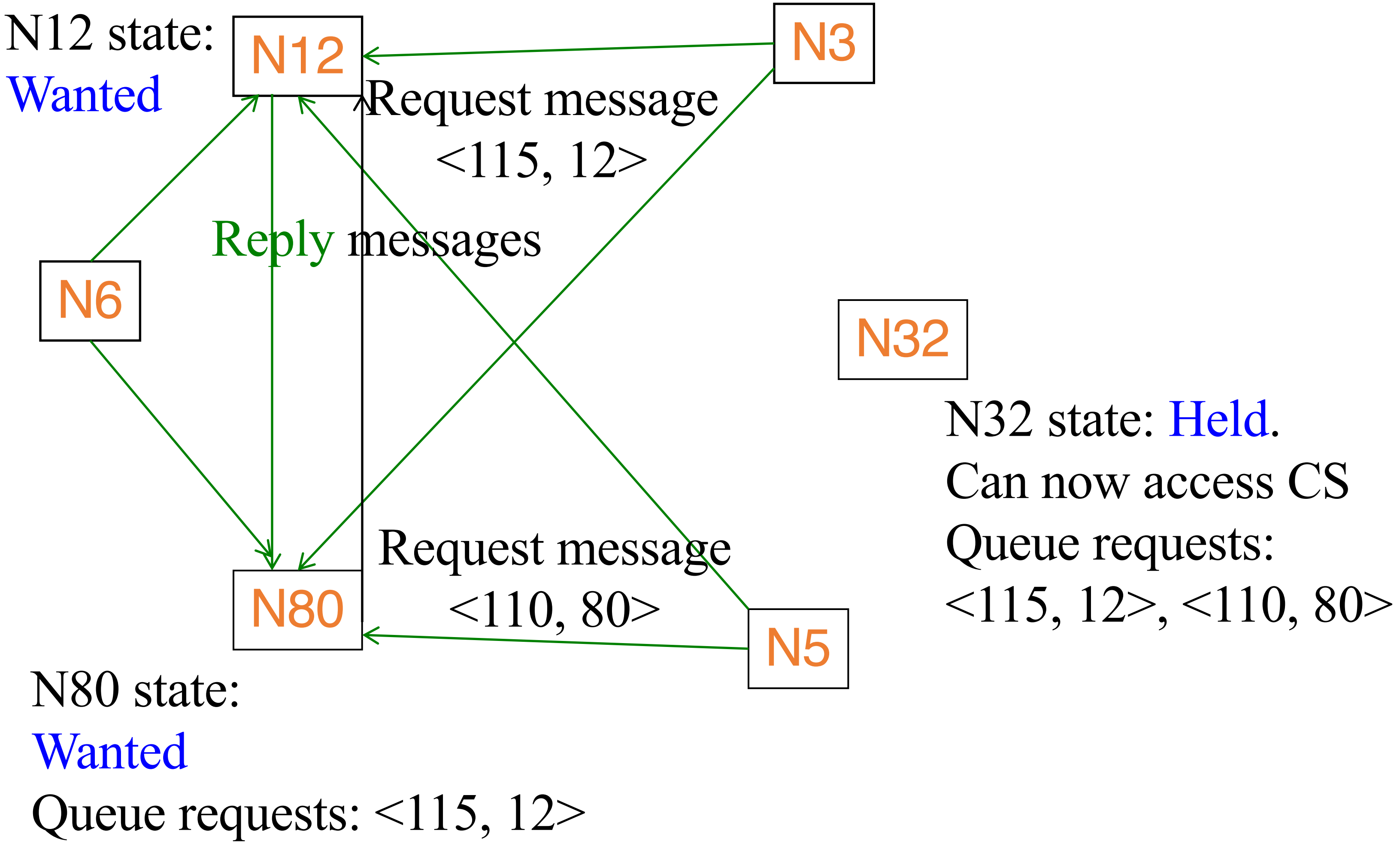
Example: Ricart-Agrawala Algorithm



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

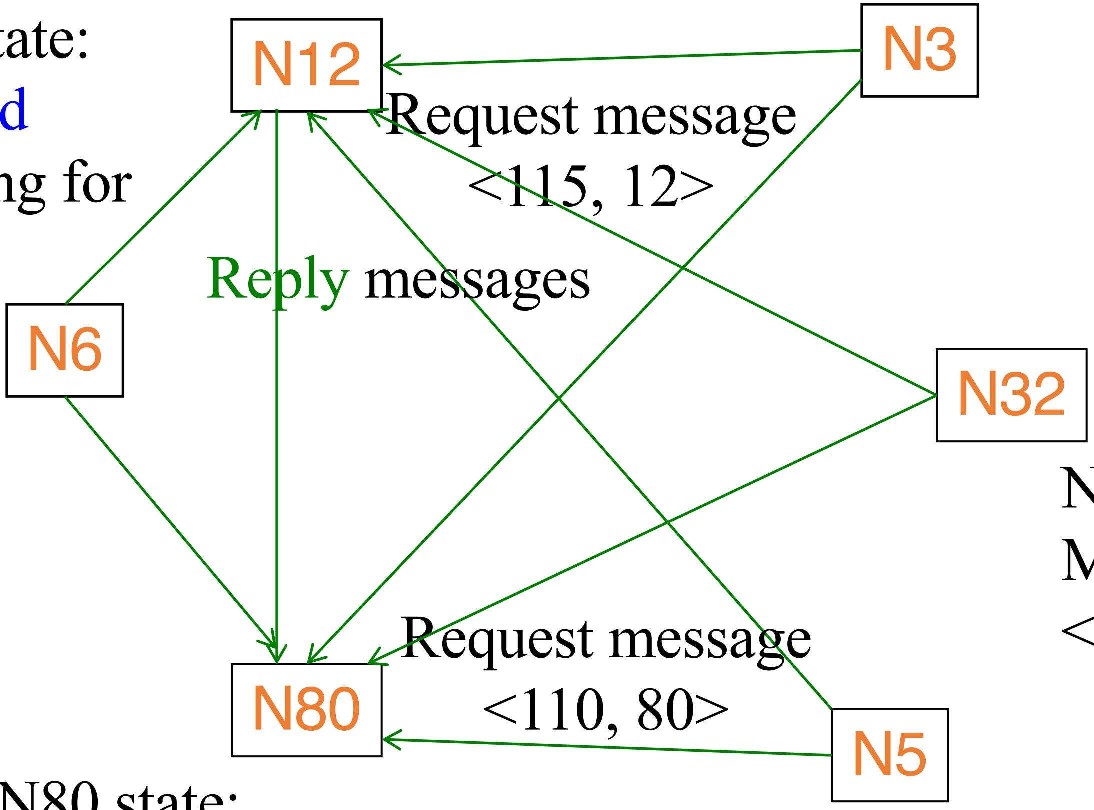
N80 state:
Wanted
Queue requests: <115, 12> (since > (110, 80))

Example: Ricart-Agrawala Algorithm



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 Algorithm

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

Analysis: Ricart-Agrawala Algorithm (cont.)

- Liveness
  - Worst-case: wait for all other (*N-1*) processes to send Reply
- Fairness
  - Requests with lower Lamport timestamps are granted earlier

Performance: Ricart-Agrawala Algorithm

- 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
- Client delay: one round-trip time
- Synchronization delay: one message transmission time

Performance: Ricart-Agrawala Algorithm

- Compared to Ring-Based approach, in Ricart-Agrawala approach
  - Client/synchronization delay has now gone down to O(1)
  - But message complexity has gone up to O(N)
- Can we get both down?

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 process one is given access to CS (Critical Section) at a time

=> A sublinear O($\sqrt{N}$) message complexity

Maekawa’s algorithm: key idea

- Each P_i is associated with a voting set V_i. Divide the set of processes into subsets that satisfy the following conditions:
  a) $i \in V_i$
  b) $V_i \cap V_j \neq \emptyset, \forall i, j$

- Main idea: Each P_i is required to receive permission from V_i only. Correctness requires that multiple processes will never receive permission from all members of their respective subsets.

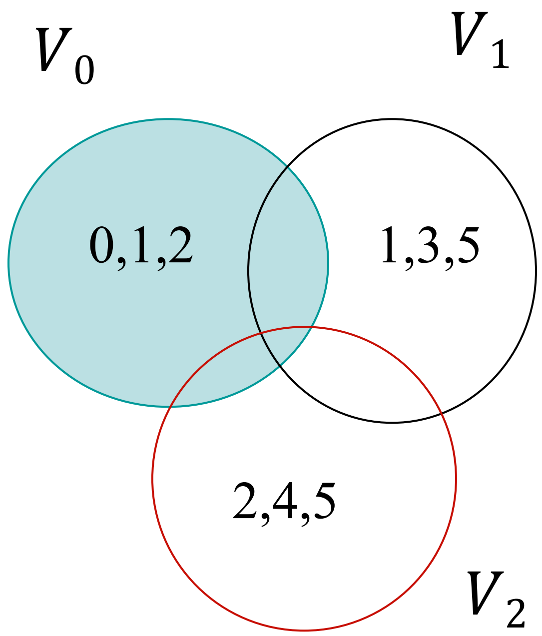
Maekawa's voting sets

- Each P_i is associated with a subset V_i. Divide the set of processes into subsets that satisfy the following conditions:
  a) $i \in V_i$
  b) $V_i \cap V_j \neq \emptyset, \forall i, j$
  c) $|V_i| = K, \forall i$
  d) Any i is contained in M $V_i's$
- Maekawa showed that $K = M \sim \sqrt{N}$ works best
- 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 $K = 2\sqrt{N} - 1$

Example: Maekawa's voting sets

Example. Let there be seven processes $0, 1, 2, 3, 4, 5, 6$

$$
\begin{aligned}
V_0 &= \{0, 1, 2\} \\
V_1 &= \{1, 3, 5\} \\
V_2 &= \{2, 4, 5\} \\
V_3 &= \{0, 3, 4\} \\
V_4 &= \{1, 4, 6\} \\
V_5 &= \{0, 5, 6\} \\
V_6 &= \{2, 3, 6\}
\end{aligned}
$$



- $K = 3, M = 3$

Maekawa: Key Differences From Ricart-Agrawala

- 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

Actions (cont.)

- 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

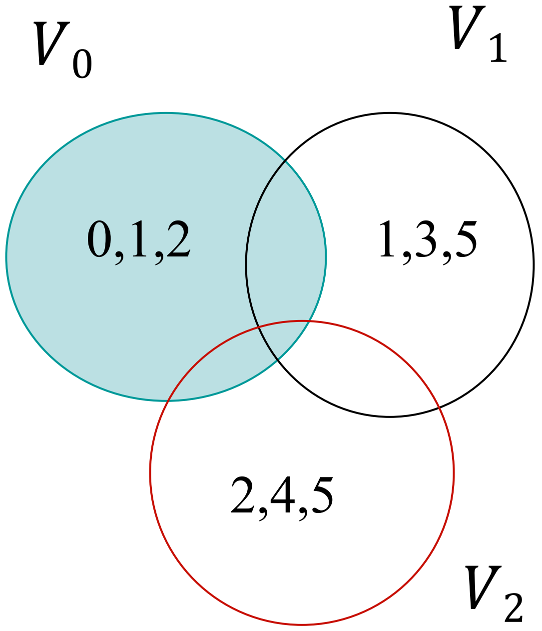
  voted = true

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*
  - ***Assume 0, 1, 2 want to enter their critical sections.***
  - From V_0= {0,1,2}, 0,2 send reply to 0, but 1 sends reply to 1;
  - From V_1= {1,3,5}, 1,3 send reply to 1, but 5 sends reply to 2;
  - From V_2= {2,4,5}, 4,5 send reply to 2, but 2 sends reply to 0;
  - Now, 0 waits for 1 (to send a release), 1 waits for 2 (to send a release), and 2 waits for 0 (to send a release). So, deadlock is possible!
- There are deadlock-free versions



Performance

- Message complexity
  - $2\sqrt{N}$ 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 => $\sqrt{N} = 1\mathrm{K}$
- Client delay: One round trip time
- Synchronization delay: 2 message transmission times

Why $\sqrt{N}$?

- Each voting set is of size *K*
- Each process belongs to *M* other voting sets
- Total number of voting set members (processes may be repeated) = *K*N*
- But since each process is in *M* voting sets
  - *K*N/M = N => K = M*   (1)

- Consider a process P_i
  - Total number of voting sets = members present in P_i's voting set and all their voting sets = *(M-1)*K + 1*
  - All processes in group must be in above
  - To minimize the overhead at each process (*K*), need each of the above members to be unique, i.e.,
    - *N = (M-1)*K + 1*
    - *N = (K-1)*K + 1* (due to (1))
    - *K* ~ $\sqrt{N}$