Distributed Systems

Monsoon 2024

Lecture 6

International Institute of Information Technology

Hyderabad, India

Mutual Exclusion

- Refer to the same topic from operating systems.
- Give some examples of mutual exclusion in systems.

Mutual Exclusion

- Refer to the same topic from operating systems.
- Broadly, mutual exclusion mandates that for a shared resource that can be used by at most one process at any time, the OS has to ensure exclusive access to such resources in addition to other properties of the solution.
  - Examples: Printer, network, a database table
- Otherwise, incorrect results may ensue.

Mutual Exclusion

- If no guarantees of mutual exclusion exist, consider the following scenario.
- A bank account is read by two different processes. The balance is Rs. 500.
- Each process wants to add Rs. 1000 to the balance.
- Both calculate the new balance to be Rs. 1500.
- If these two updates do not happen in exclusive manner, the new balance may be incorrect.
  - The new balance can be for instance, Rs. 1500. Whereas it should be Rs. 2500.

Mutual Exclusion

- Mutual exclusion in operating systems solved by algorithms such as:

Mutual Exclusion

- Mutual exclusion in operating systems solved by algorithms such as:
  - The bakery algorithm
  - Peterson’s algorithm
- What do the above algorithms assume?
- Review this material for your own ease of understanding.

Mutual Exclusion

- Mutual exclusion in operating systems solved by algorithms such as
  - The bakery algorithm
  - Peterson’s algorithm
- These algorithms require assumptions on the system such as
  - Centralized control
  - Atomic operations
- Review this material for your own ease of understanding.

Mutual Exclusion

- Mutual exclusion is important in the context of distributed computations too.
- However, achieving mutual exclusion may not be any easier than in centralized settings.
    - No centralized control!!
- Surprisingly, several algorithms exist in this setting.
- We will study some of them in this lecture.

Mutual Exclusion

- Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
- Only one process is allowed to execute the critical section (CS) at any given time.
- In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.
- Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.

Mutual Exclusion

- Requirements of Mutual Exclusion Algorithms
  1. Safety: At any instant, only one process can execute the critical section.
  2. Liveness: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
  3. Fairness: Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

Lamport's Algorithm for Mutual Exclusion

- Lamport used his logical scalar clocks and FIFO assumption on message delivery to design an algorithm for mutual exclusion.
- Also assume a bidirectional channel between each pair of processors.
- Every site S_i keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps.

Lamport's Algorithm for Mutual Exclusion

- Requesting the critical section:
  - REQUEST(ts_i , i) : When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i , i) message to all other sites and places the request on request_queue$_i$.
    - ((ts_i , i) denotes the timestamp of the request.)
  - REPLY: When a site S_j receives the REQUEST(ts_i, i) message from site S_i, places site S_i 's request on request_queue$_j$ and it returns a timestamped REPLY message to S_i.

Lamport's Algorithm for Mutual Exclusion

- Executing the critical section:
  - Two conditions: Site S_i enters the CS when the following two conditions hold:
    - L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

Lamport's Algorithm for Mutual Exclusion

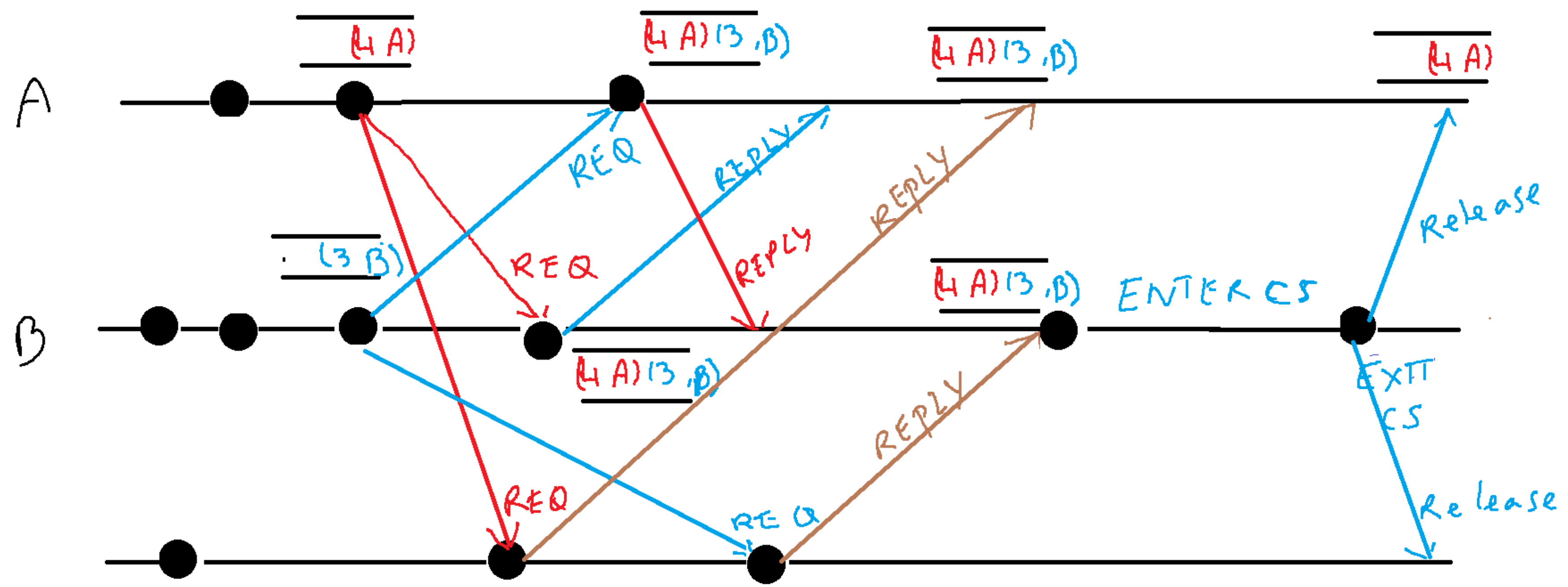
- Executing the critical section:
  - Two conditions: Site S_i enters the CS when the following two conditions hold:
    - L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.
    - L2: S_i 's request is at the top of $request\_queue_i$.

Lamport's Algorithm

- RELEASE:
  - Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.

- When a site S_j receives a RELEASE message from site S_i, it removes S_i's 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 the CS.

Example Run – Lamport's Algorithm



Lamport's Algorithm

- The queues maintained at each process is like a shared priority queue.
- However, not all queues may have the same contents at all times.
- Messages can have unpredictable delays leading to lack of consistent information across the system.

Lamport's Algorithm

- Theorem: Lamport's algorithm achieves mutual exclusion.
- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently.
- For this to happen conditions L1 and L2 must hold at both the sites concurrently.

Lamport's Algorithm

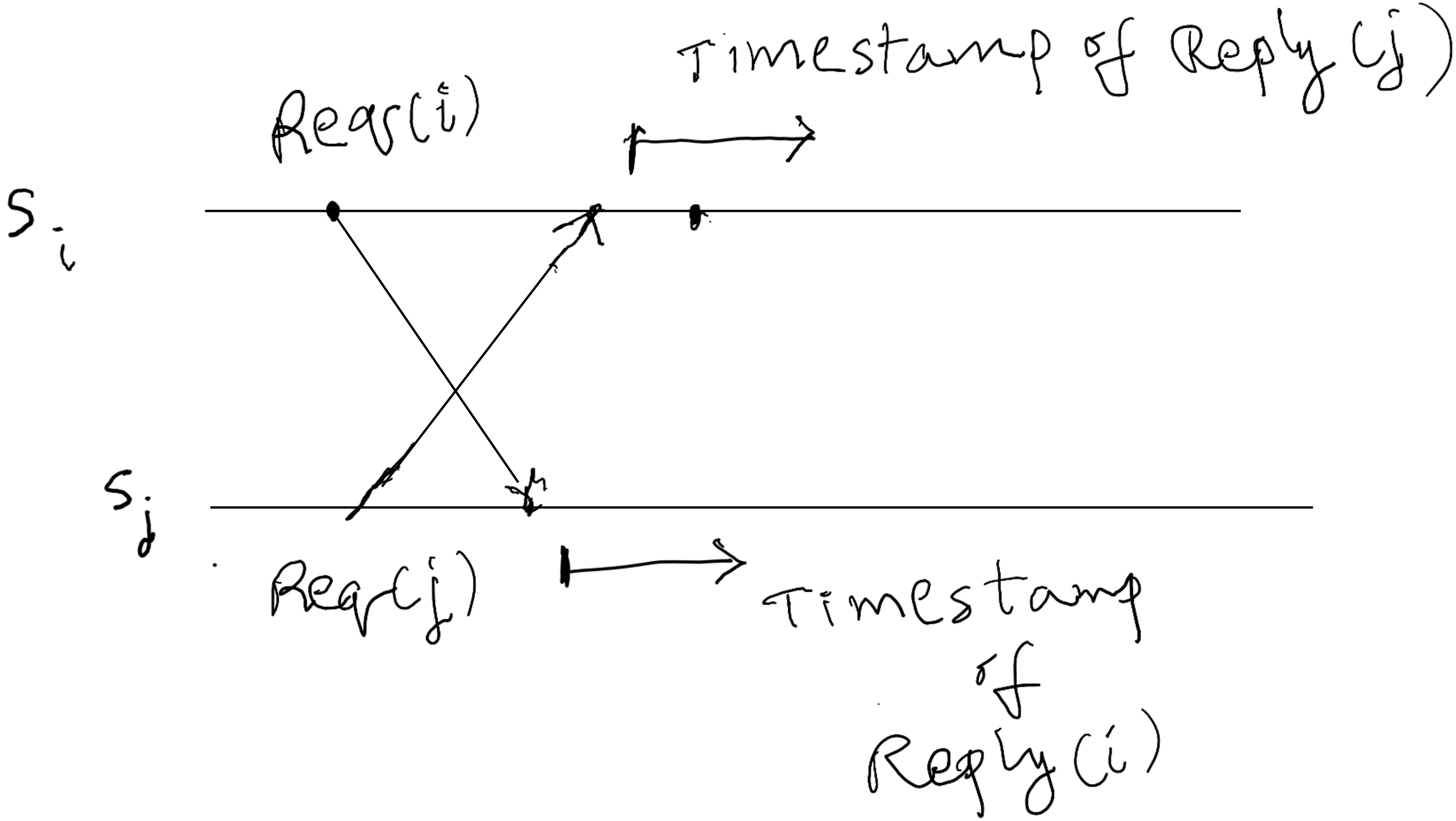
- Theorem: Lamport's algorithm achieves safety/ mutual exclusion.
- This implies that at some instant in time, say t, both L1 and L2 hold at both S_i and S_j.
  - By L2, S_i and S_j have their own requests at the top of their request queues.
- Without loss of generality, assume that S_i 's request has smaller timestamp than the request of S_j.
    - Note that smaller means that $(ts_i, i) < (ts_j, j)$,

Lamport's Algorithm

- Theorem: Lamport's algorithm achieves safety/ mutual exclusion.
- By L2, S_i and S_j have their own requests at the top of their request queues.
- Without loss of generality, assume that S_i 's request has smaller timestamp than the request of S_j.
- It is clear that at instant t the request of S_i must be present in $request\_queue_j$ when S_j was executing its CS. WHY?

Lamport's Algorithm

- Theorem: Lamport's algorithm achieves safety/ mutual exclusion.
- By L2, S_i and S_j have their own requests at the top of their request queues.
- Without loss of generality, assume that S_i 's request has smaller timestamp than the request of S_j.
- It is clear that at instant t the request of S_i must be present in request_queue$_j$ when S_j was executing its CS. WHY?
- This implies that S_j 's own request is at the top of its own request queue when a smaller timestamp request, S_i 's request, is present in the request_queue$_j$ – a contradiction!



At S_j, Recv(Req(i)) < Recv(Reply(j))

Lamport's Algorithm

- Theorem: Lamport's algorithm is fair.
- The proof is by contradiction. Suppose a site S_i 's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i.
- For S_j to execute the CS, it has to satisfy the conditions L1 and L2.
- This implies that at some instant in time, say t, S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.

Lamport's Algorithm

- Theorem: Lamport's algorithm is fair.
- But the request queue at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So S_i 's request must be placed ahead of the S_j 's request in the $request\_queue_j$.
- This is a contradiction.

Further Reading

- Read about the number of messages exchanged in Lamport’s algorithm.
- Read about a couple of optimizations to reduce the number of messages.

Fairness of Lamport's Algorithm

- Lamport's algorithm requires $3(N - 1)$ messages per CS grant.
- Some optimizations exist to reduce the number of messages to 2(N-1) in some cases.

Ricart-Agarwala's Algorithm

- Removes the assumption on FIFO delivery guarantee of channels.
- Uses two types of messages
- A process sends a REQUEST message to all other processes to request their permission to enter the critical section.
- A process sends a REPLY message to a process to grant its permission to that process.

Ricart-Agarwala's Algorithm

- Processes use Lamport's bakery algorithm style priority numbers to assign a priority to critical section requests.
- Each process p_i maintains a Boolean Request_Deferred array, RD_i, the size of which is the same as the number of processes in the system.
- Initially, $Rd_i[j]=0$ for all i and j. Whenever p_i defers the request sent by p_j, it sets $Rd_i[j]=1$ and after it has sent a REPLY message to p_j, it sets $RD_i[j]=0$.

Ricart-Agarwala’s Algorithm

- Processes use Lamport’s bakery algorithm style priority numbers to assign a priority to critical section requests.
- Entry to the critical section is in the order of the priority numbers.
- Priority numbers are maintained in a distributed manner.
  - Means that processes may not exactly know what is the largest number seen so far.
- Each process keeps track of the largest priority number it has heard of so far.

Ricart-Agarwala's Algorithm

- Requesting the critical section:
  1. When a site S_i wants to enter the CS, it broadcasts a REQUEST along with the priority number message to all other sites.
  2. When site S_j receives a REQUEST message from site S_i,
     a) it sends a REPLY message to site S_i if
        i. S_j not interested: The site S_j is neither requesting nor executing the CS, or
        ii. S_j is interested and (S_i < S_j) : The site S_j is requesting, and S_i's request's priority number is smaller than site S_j's own request's timestamp.
     b) Otherwise, the reply is deferred (delayed).
        i. To note that a reply is delayed, site S_j sets $RD_j[i]=1$.

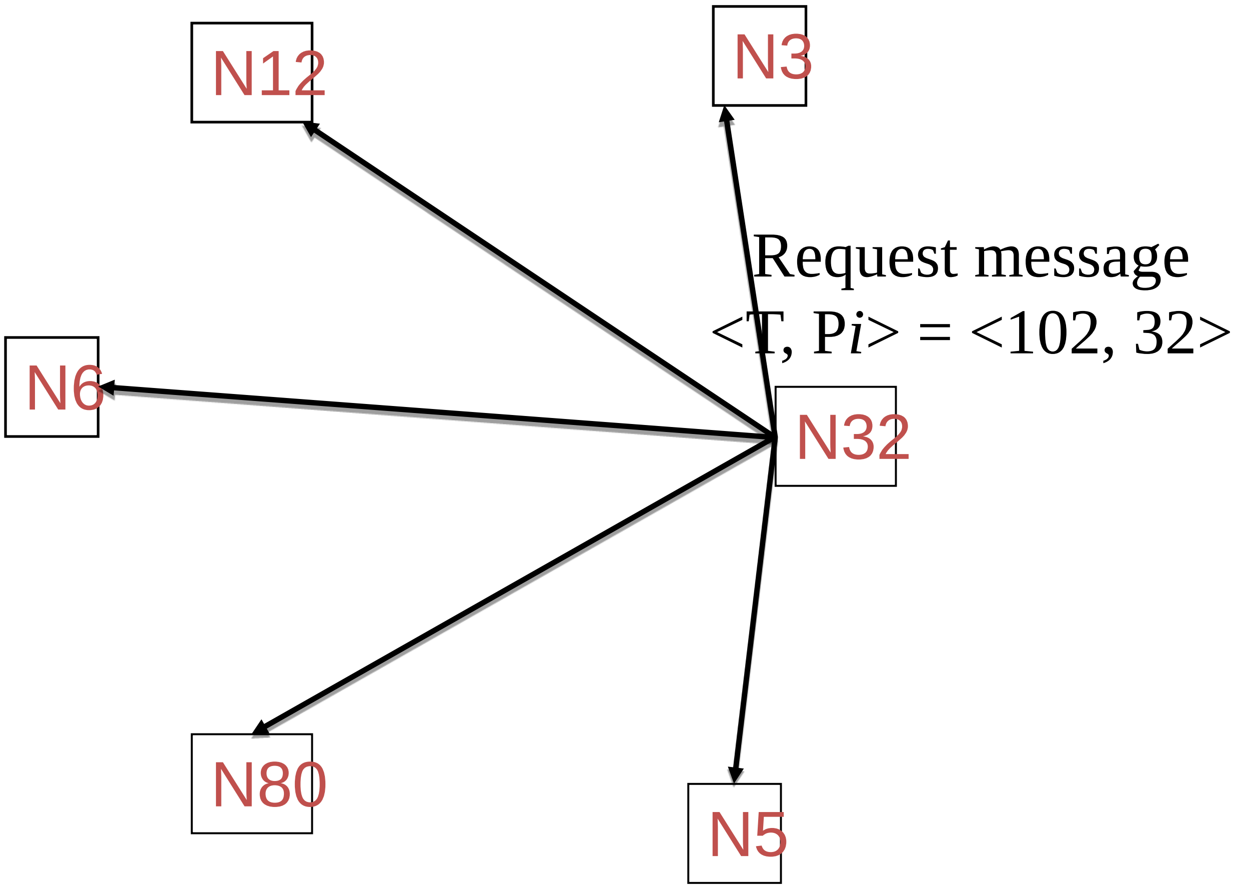
Ricart-Agarwala's Algorithm

- Executing the critical section:
  - Site S_i enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

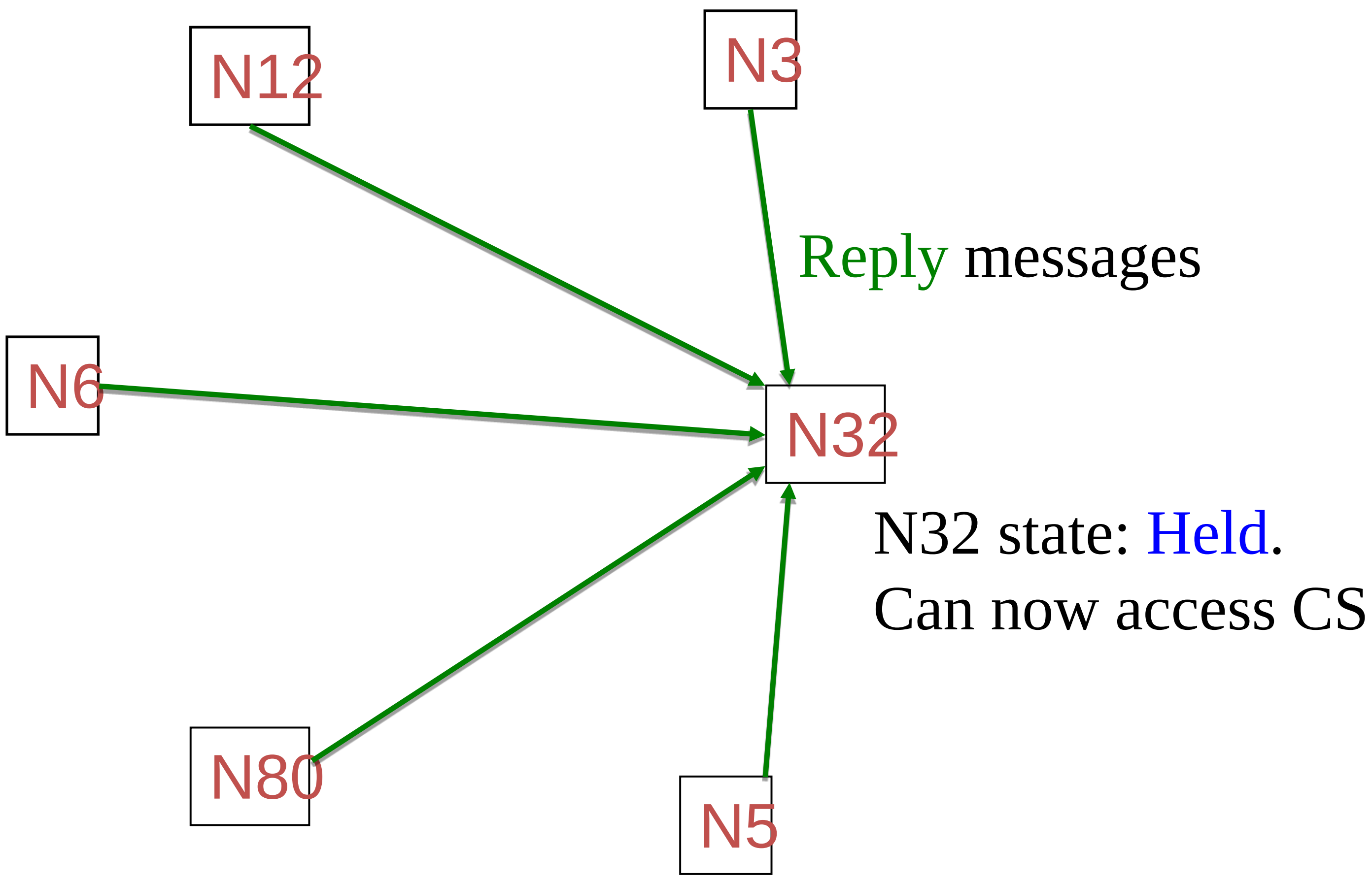
Ricart-Agarwala's Algorithm

- Releasing the critical section:
  - When site S_i exits the CS, it sends all the deferred REPLY messages:
    - for all j, if $RD_i[j]=1$, then send a REPLY message to S_j and set $RD_i[j]=0$.

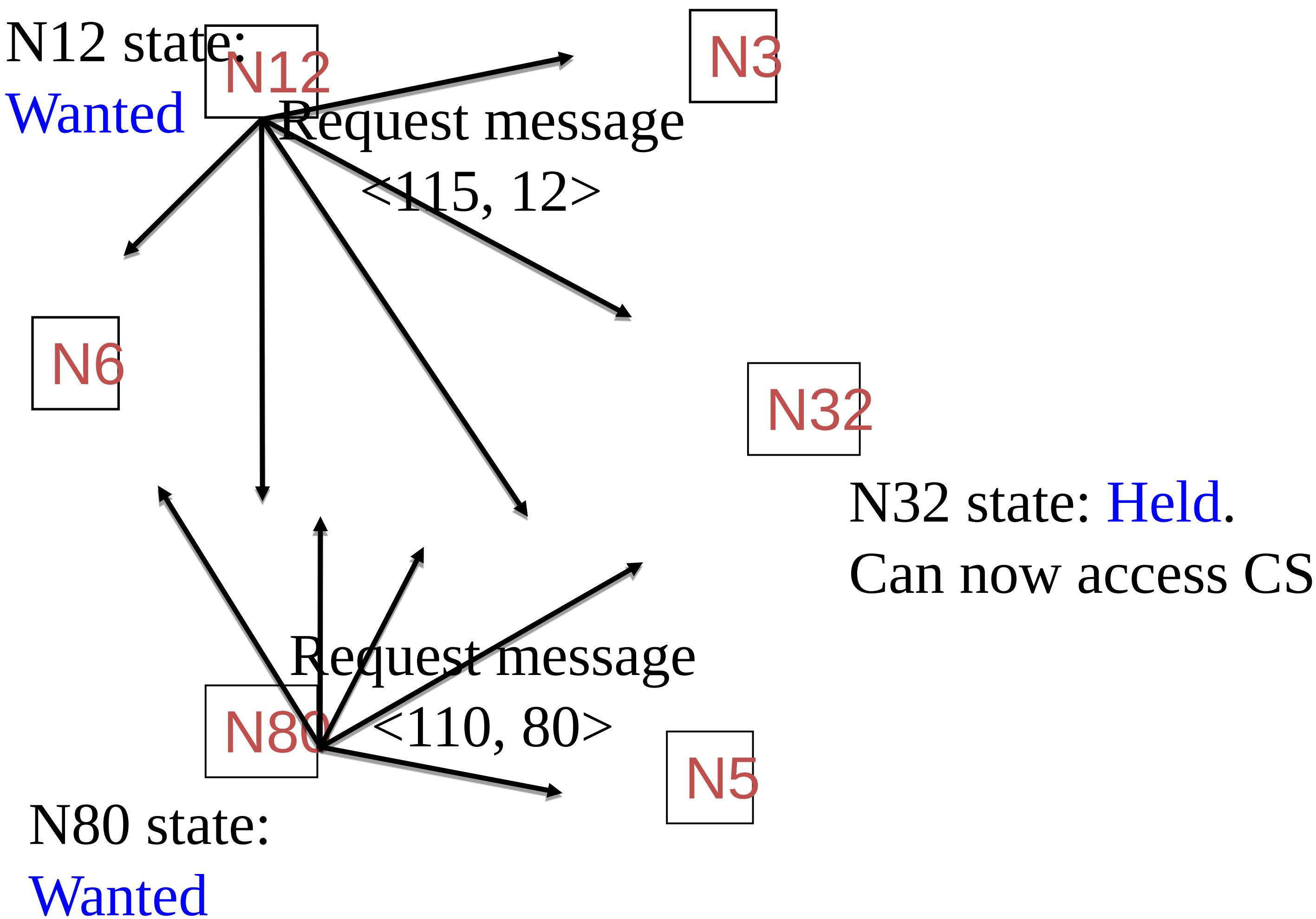
Example: Ricart-Agrawala Algorithm



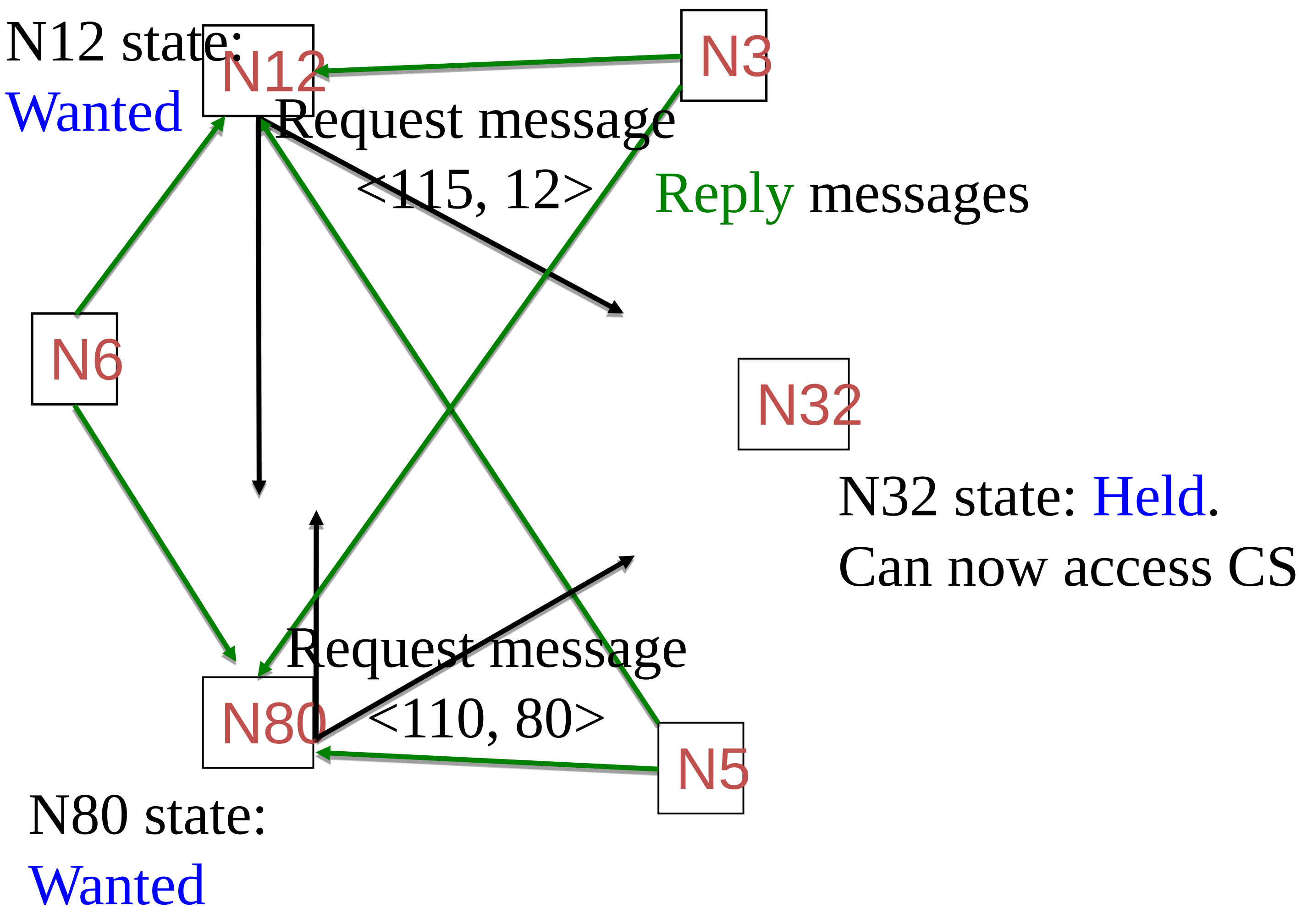
Example: Ricart-Agrawala Algorithm



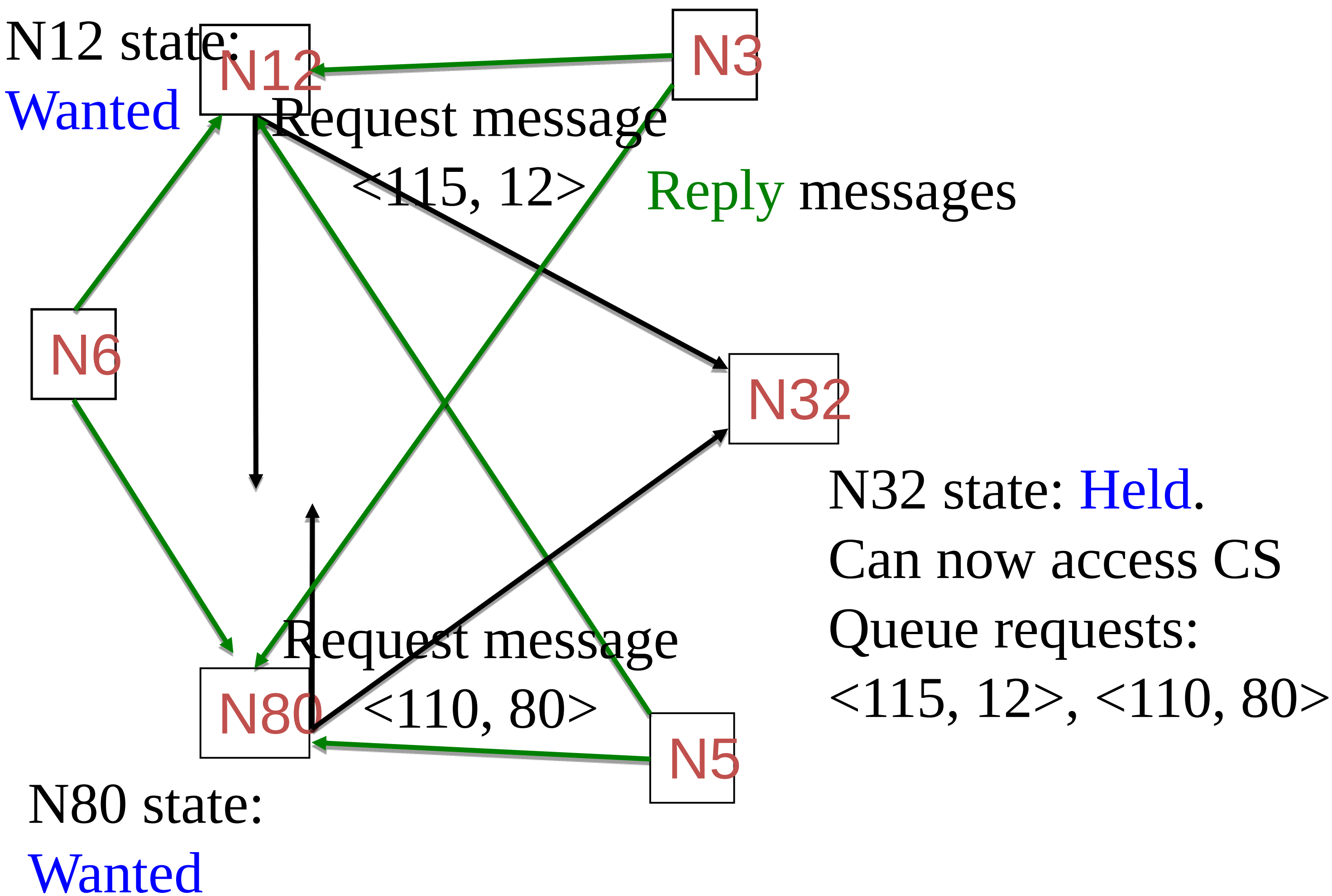
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm

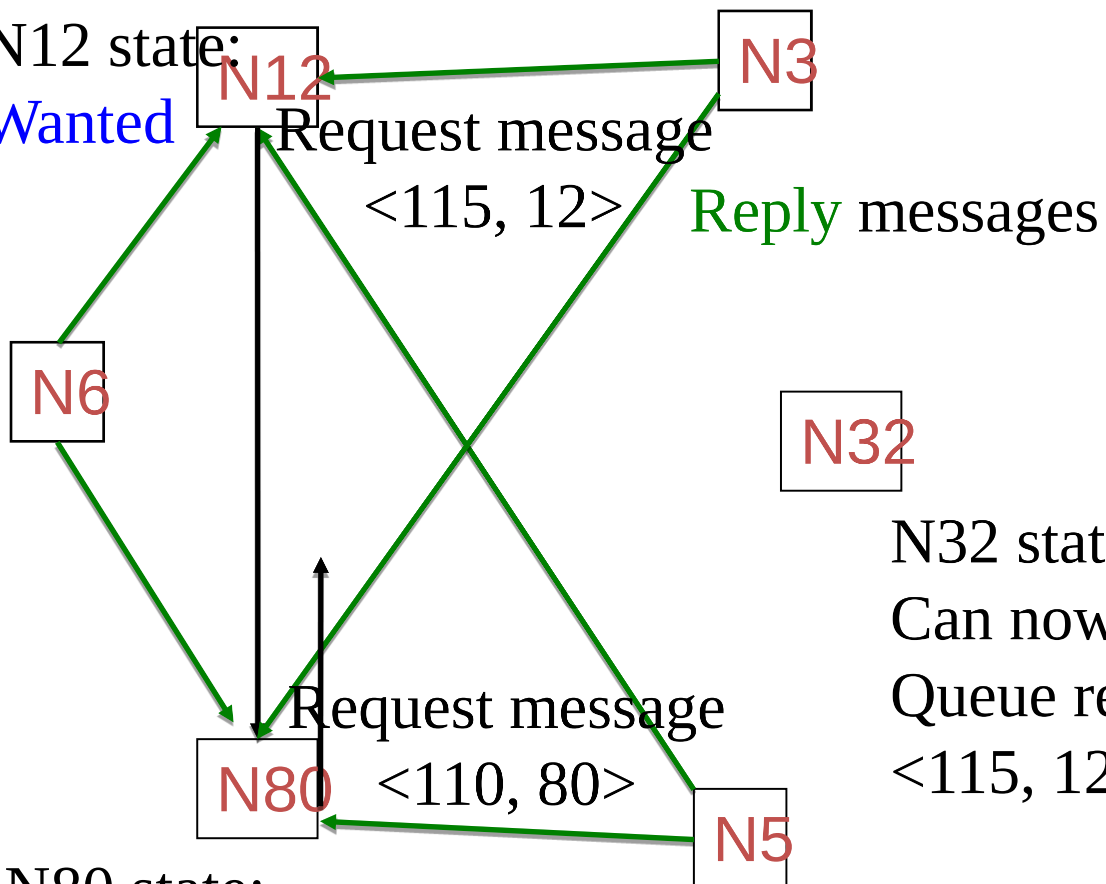


Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm

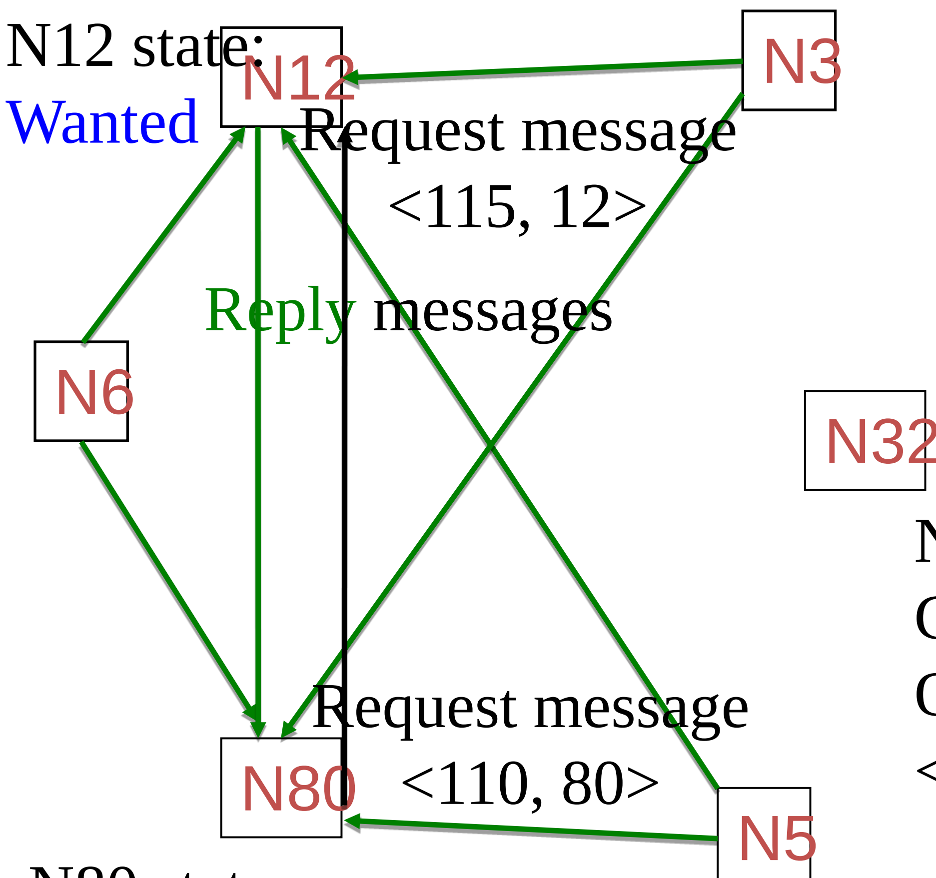
N12 state:
Wanted



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

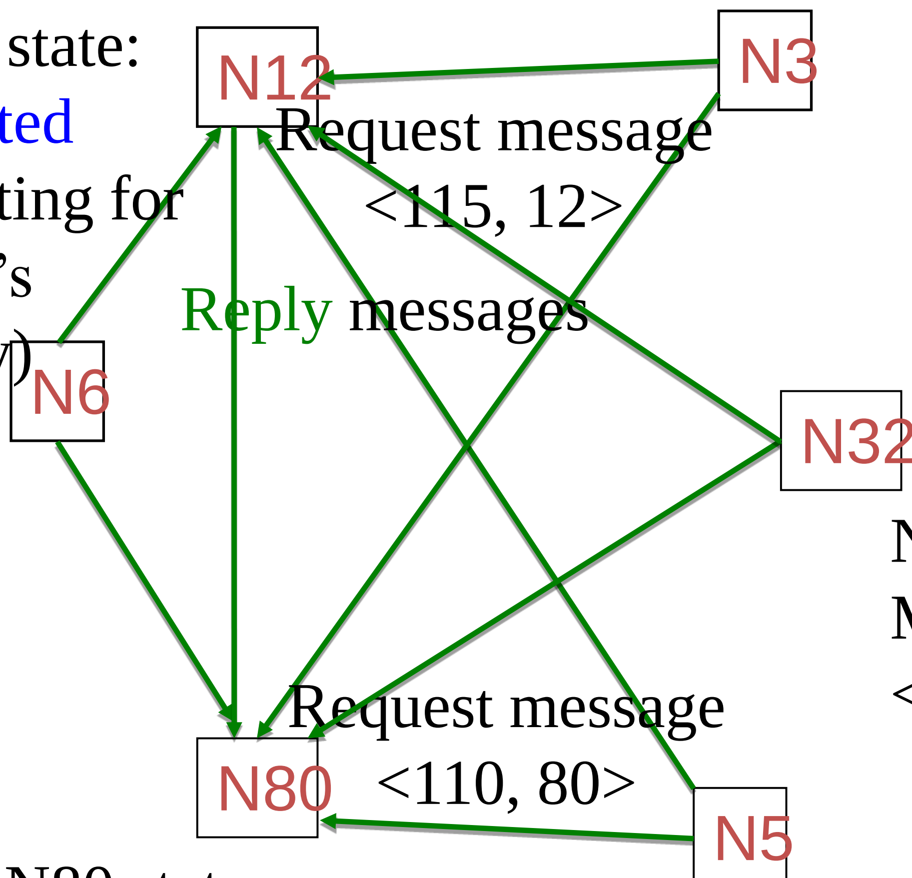


N32 state: Held.
Can now access CS
Queue requests:
<115, 12>, <110, 80>

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>

Ricart-Agarwala's Algorithm

- Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.
- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently and S_i's request has higher priority than the request of S_j.
- Clearly, S_i received S_j's request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j's request) before S_i exits the CS.
- However, this is impossible because S_j's request has lower priority.
- Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

Ricart-Agarwala’s Algorithm

- For each CS execution, Ricart-Agrawala algorithm requires (N − 1) REQUEST messages and (N − 1) REPLY messages.
- Thus, it requires 2(N − 1) messages per CS execution.