Distributed Systems

CS425/ECE428

Feb 17 2022

Instructor: Radhika Mittal

Acknowledgements for some of the materials: Indy Gupta and Nikita Borisov

Today's agenda

- **Mutual Exclusion**
  - Chapter 15.2
- **Leader Election** (if time)
  - Chapter 15.3

Recap: 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

Recap: Mutual exclusion in distributed systems

- Processes communicating by passing messages.
- Cannot share variables like semaphores!
- *How do we support mutual exclusion in a distributed system?*

Recap: Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm

Recap: System Model

- Each pair of processes is connected by reliable channels (such as TCP).
- Messages sent on a channel are eventually delivered to recipient, and in FIFO (First In First Out) order.
- Processes do not fail.
    - Fault-tolerant variants exist in literature.

Recap: Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - **Central server algorithm**
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm

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
- Ordering:
  - FIFO ordering guaranteed in order of requests received at leader
  - Not in the order in which requests were sent or the order in w~~hich processes enter CS~~!

    processes call "enter()"

Analyzing Performance

Three metrics:

- **Bandwidth**: 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 CS, or waiting)
  - *We will focus on the client delay for 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). Measure of the *throughput* of the system.

Analysis of Central Algorithm

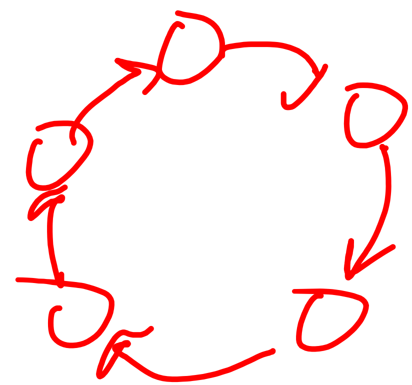
- **Bandwidth**: the total number of messages sent in each *enter* and *exit* operation.
  - 2 messages for enter
  - 1 message for exit
- **Client delay**: the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
  - 2 message latencies or 1 round-trip (request + grant) on enter.
- **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)

Limitations of Central Algorithm

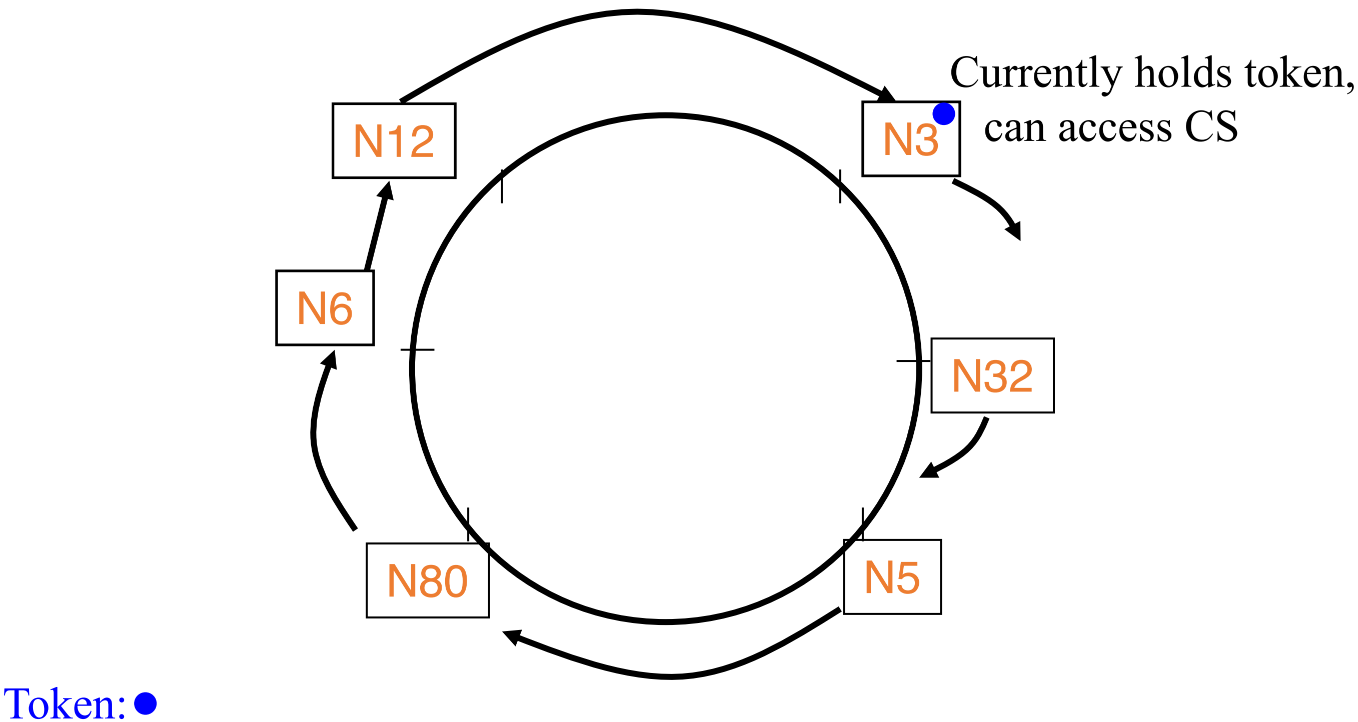
- The leader is the performance bottleneck and single point of failure.

Mutual exclusion in distributed systems

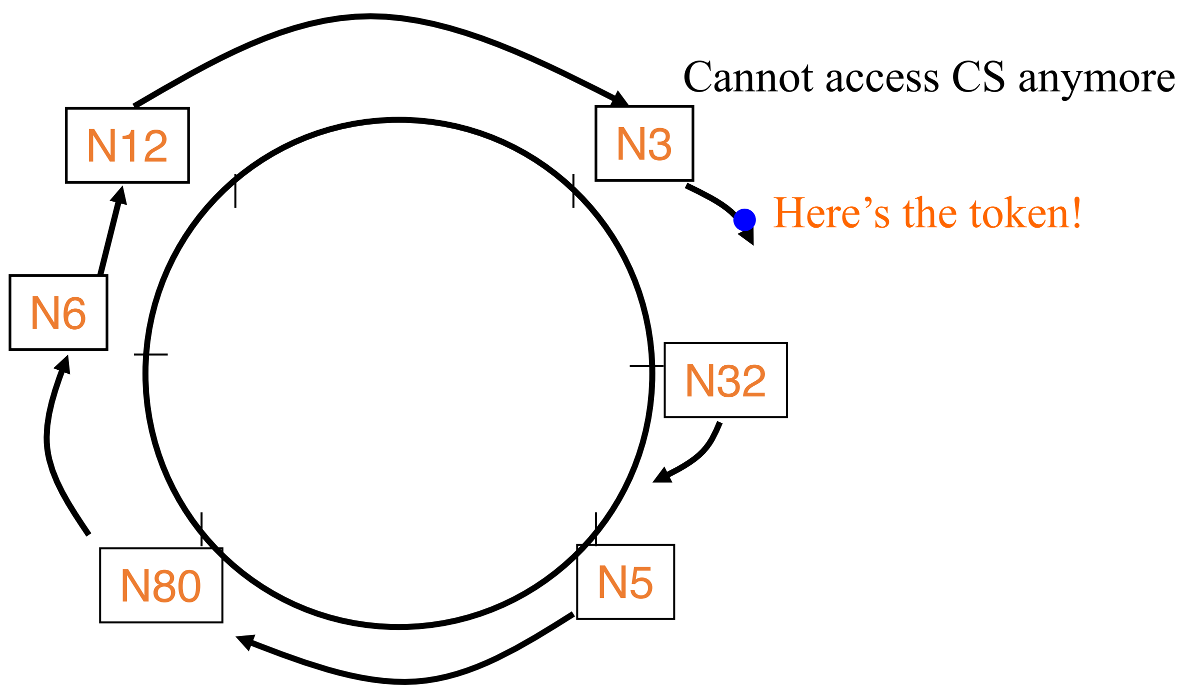
- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm



Ring-based Mutual Exclusion

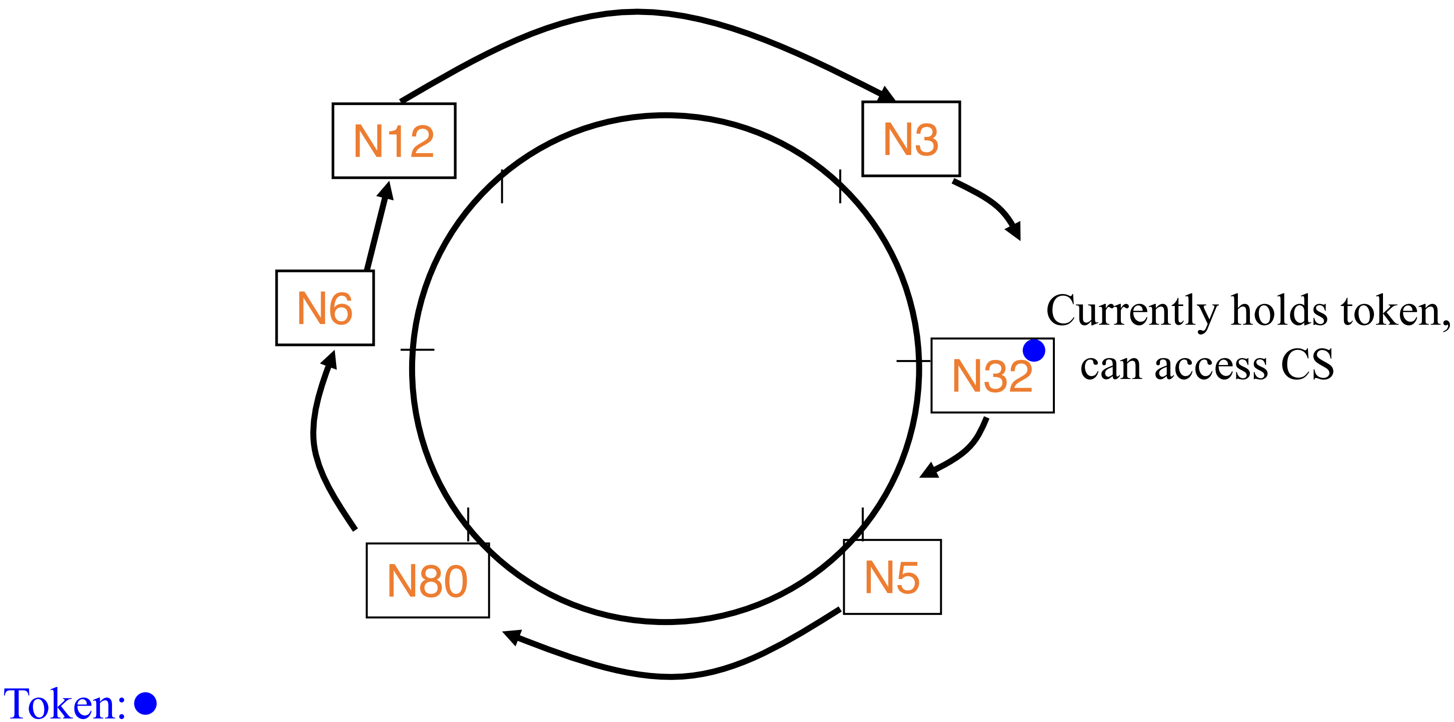


Ring-based Mutual Exclusion



Token: ●

Ring-based Mutual Exclusion



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 algorithm

- Safety
  - Exactly one token
- Liveness
  - Token eventually loops around ring and reaches requesting process (we assume no failures)
- Ordering
  - Token not always obtained in order of enter events.

Analysis of Ring-based algorithm

- Safety
  - Exactly one token
- Liveness
  - Token eventually loops around ring and reaches requesting process (we assume no failures)
- Ordering
  - Token not always obtained in order of enter events.

Analysis of Ring-based algorithm

- Bandwidth
  - Per enter, 1 message at requesting process but up to *N* messages throughout system.
  - 1 message sent per exit.
  - *Constantly consumes bandwidth even when no process requires entry to the critical section (except when a process is executing critical section).*

Analysis of Ring-based algorithm

- Client delay:
  - Best case: just received token
  - Worst case: just sent token to neighbor
  - 0 to *N* message transmissions after entering enter()
- Synchronization delay between one process' exit() from the CS and the next process' enter():
  - Best case: process in enter() is successor of process in exit()
  - Worst case: process in enter() is predecessor of process in exit()
  - Between 1 and (*N-1*) message transmissions.
- *Can we improve upon this O(n) client and synchronization delays?*

Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
    - Central server algorithm
    - Ring-based algorithm
    - Ricart-Agrawala Algorithm
    - Maekawa Algorithm

Ricart-Agrawala's 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.

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
- Requests are granted in order of causality.
- <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

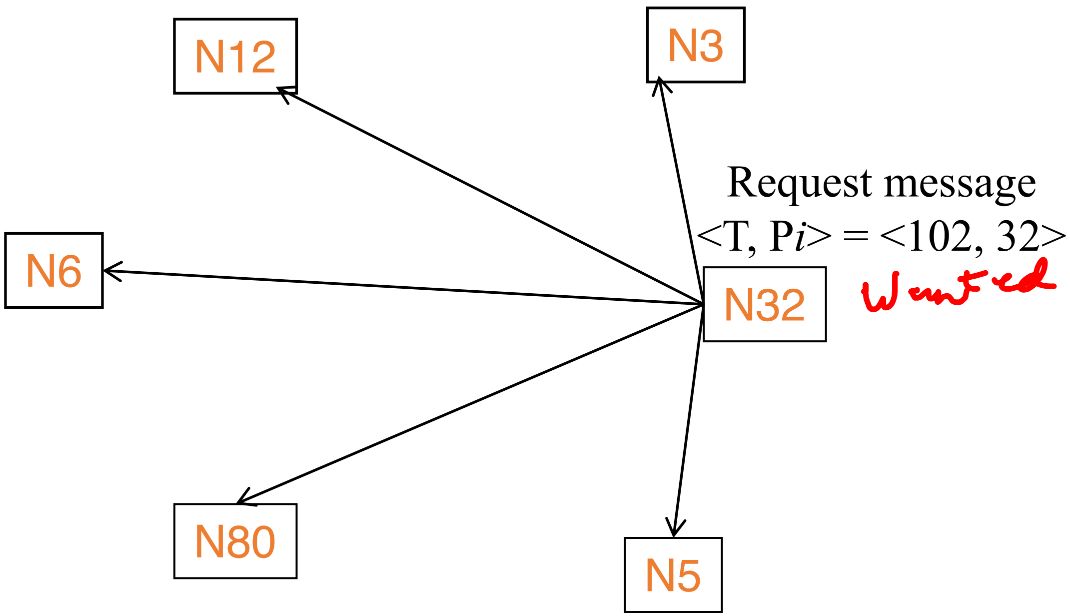
- enter() at process Pi
  - set state to <u>Wanted</u>
  - multicast "Request" <Ti, Pi> to all other processes, where Ti = current Lamport timestamp at Pi
  - wait until <u>all</u> other processes send back "Reply"
  - change state to <u>Held</u> and enter the CS
- On receipt of a Request <Tj, j> at Pi (i ≠ j):
  - if (state = <u>Held</u>) or (state = <u>Wanted</u> & (Ti, i) < (Tj, j))

    // lexicographic ordering in (Tj, j), Ti is Lamport timestamp of Pi's request

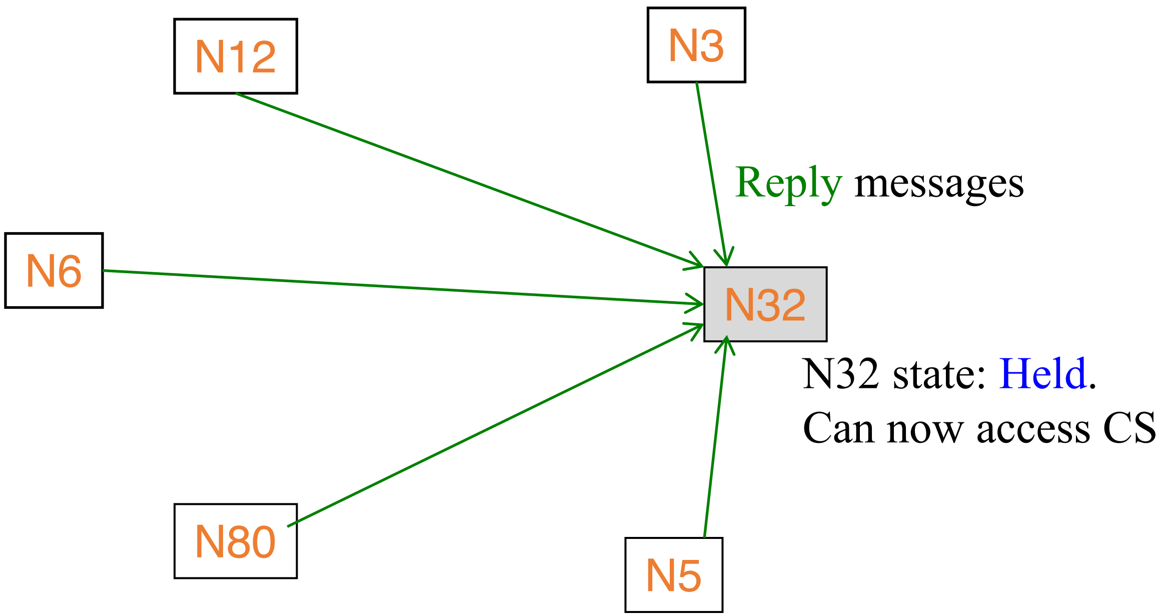
    add request to local queue (of waiting requests)

    else send "Reply" to Pj
- exit() at process Pi
  - change state to <u>Released</u> and "Reply" to <u>all</u> locally queued requests.

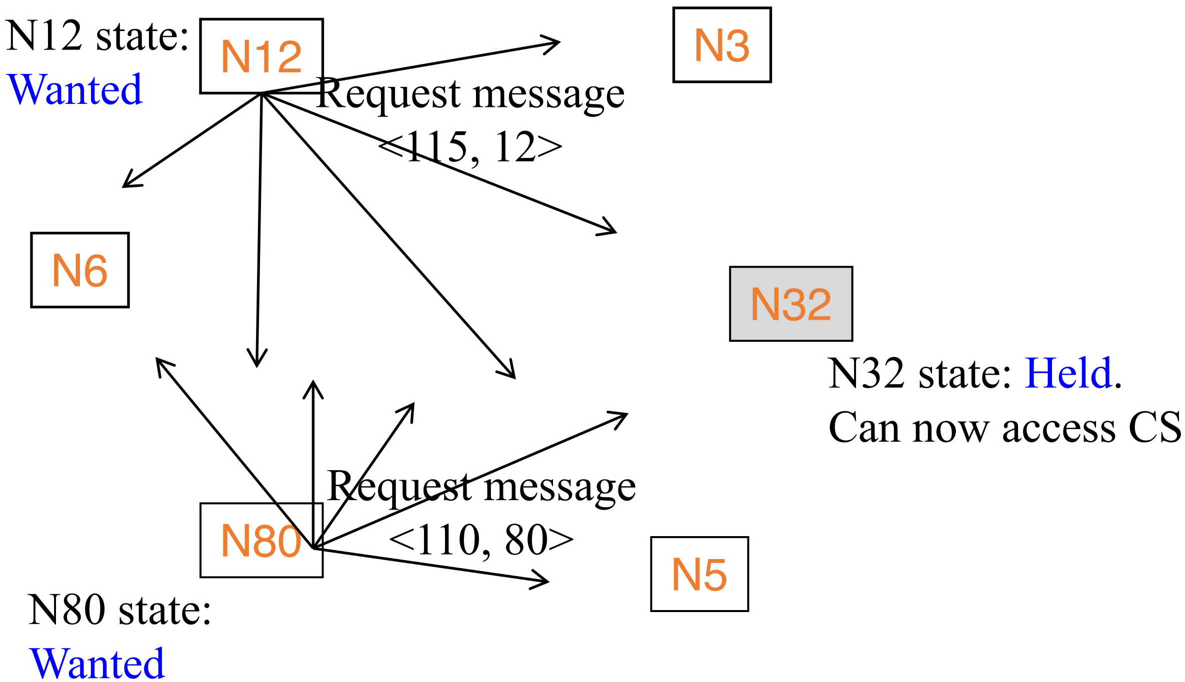
Example: Ricart-Agrawala Algorithm



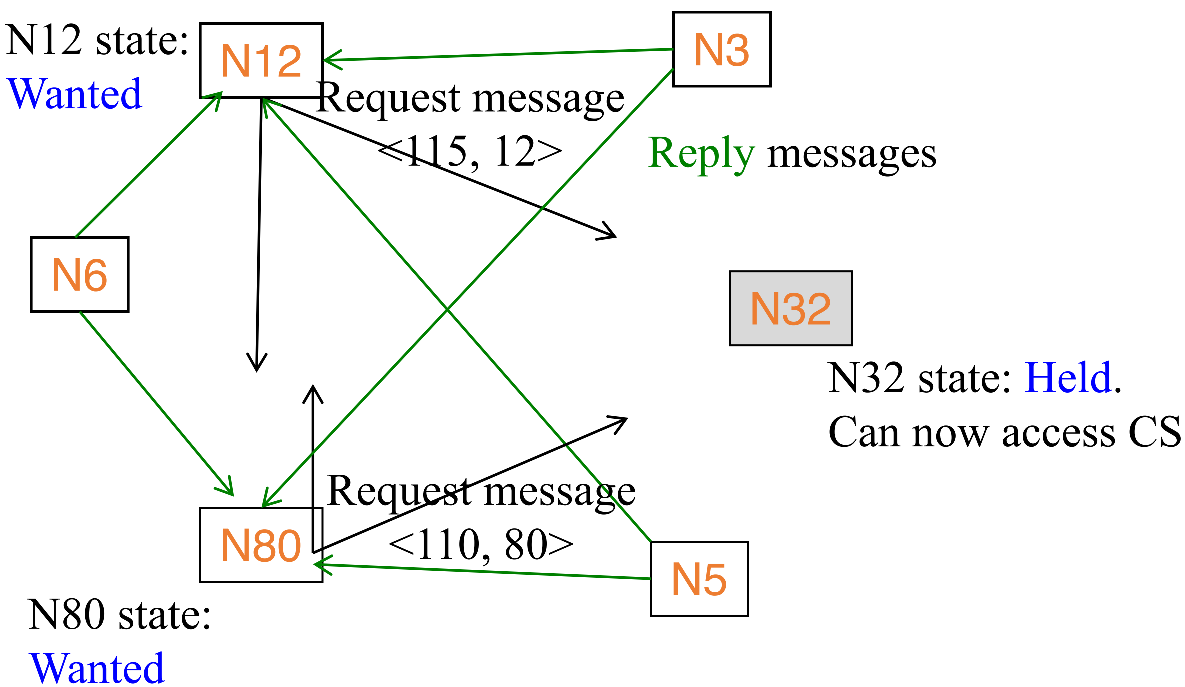
Example: Ricart-Agrawala Algorithm



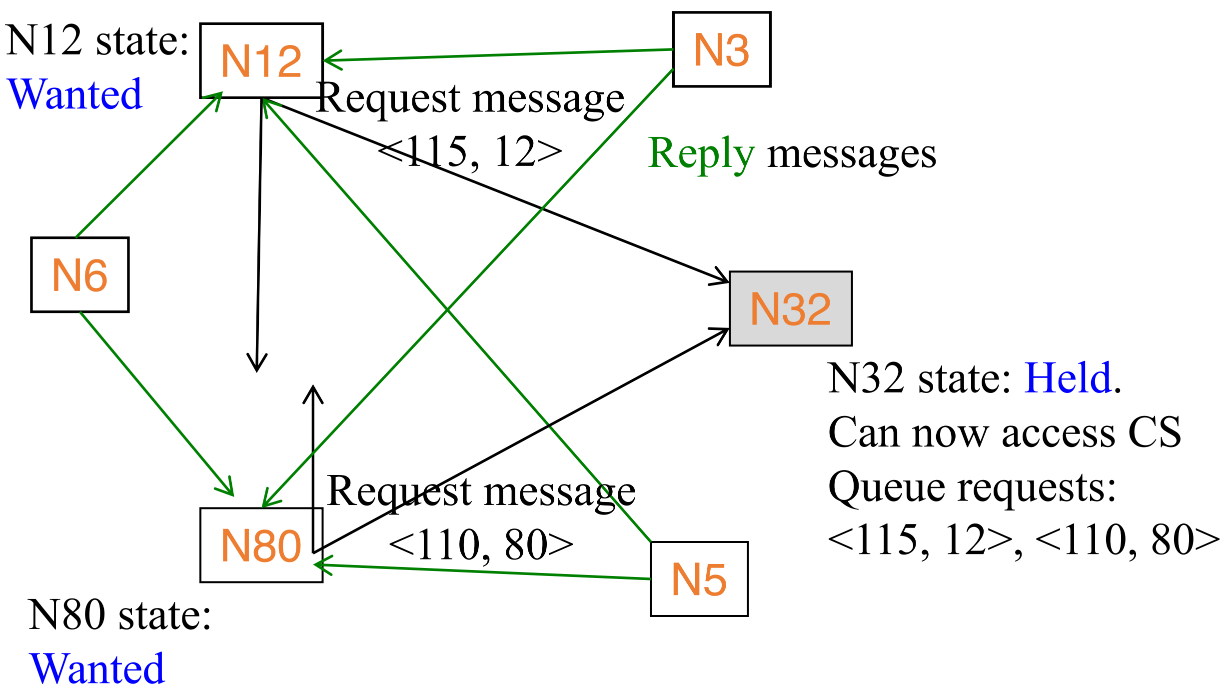
Example: Ricart-Agrawala Algorithm



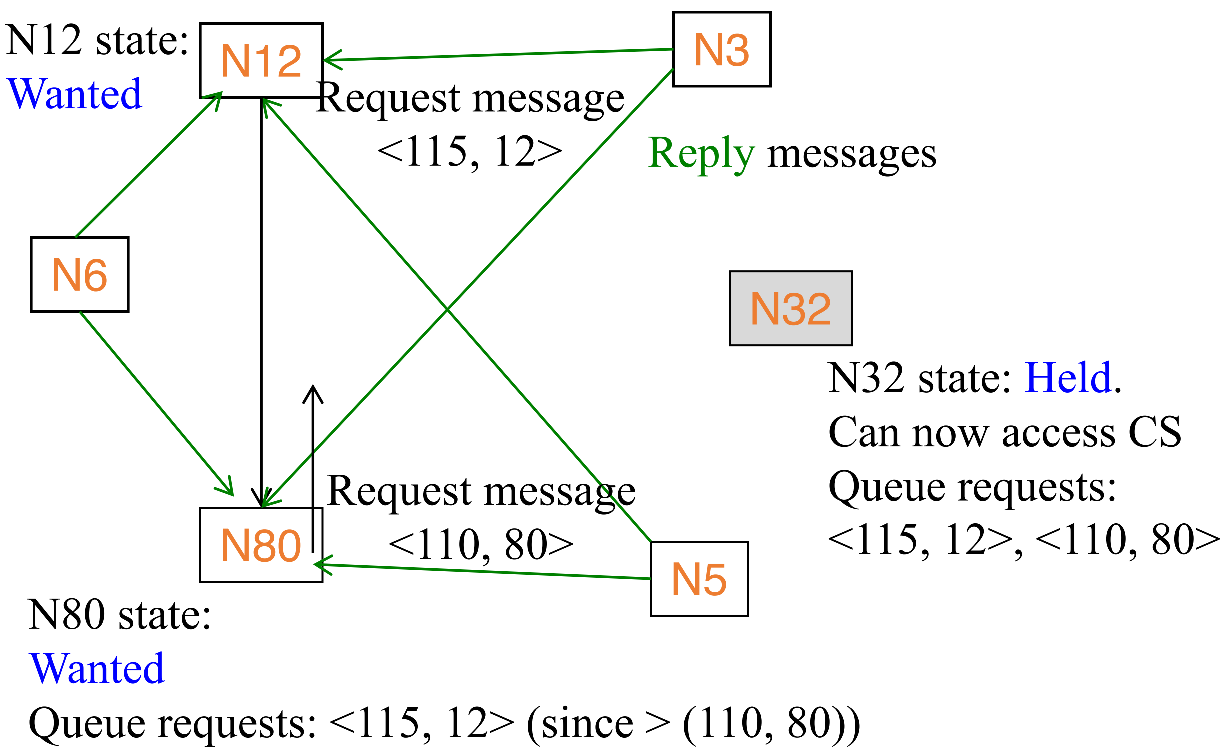
Example: Ricart-Agrawala Algorithm



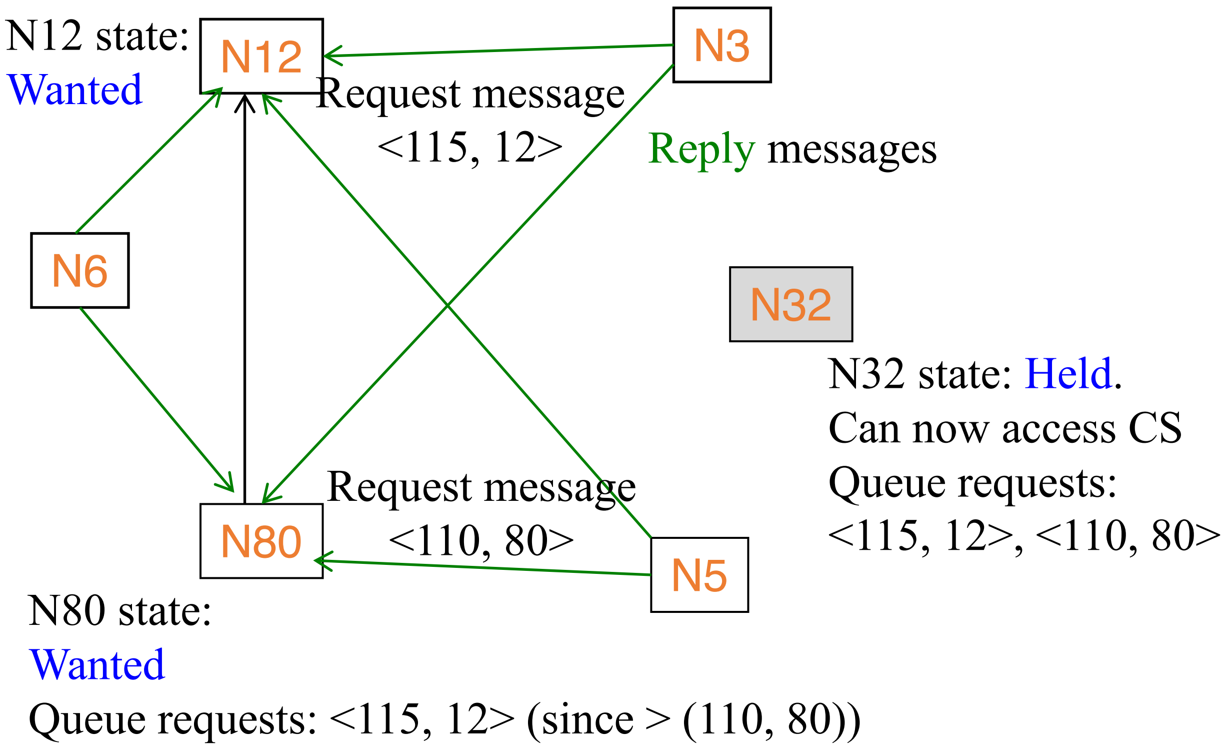
Example: Ricart-Agrawala Algorithm



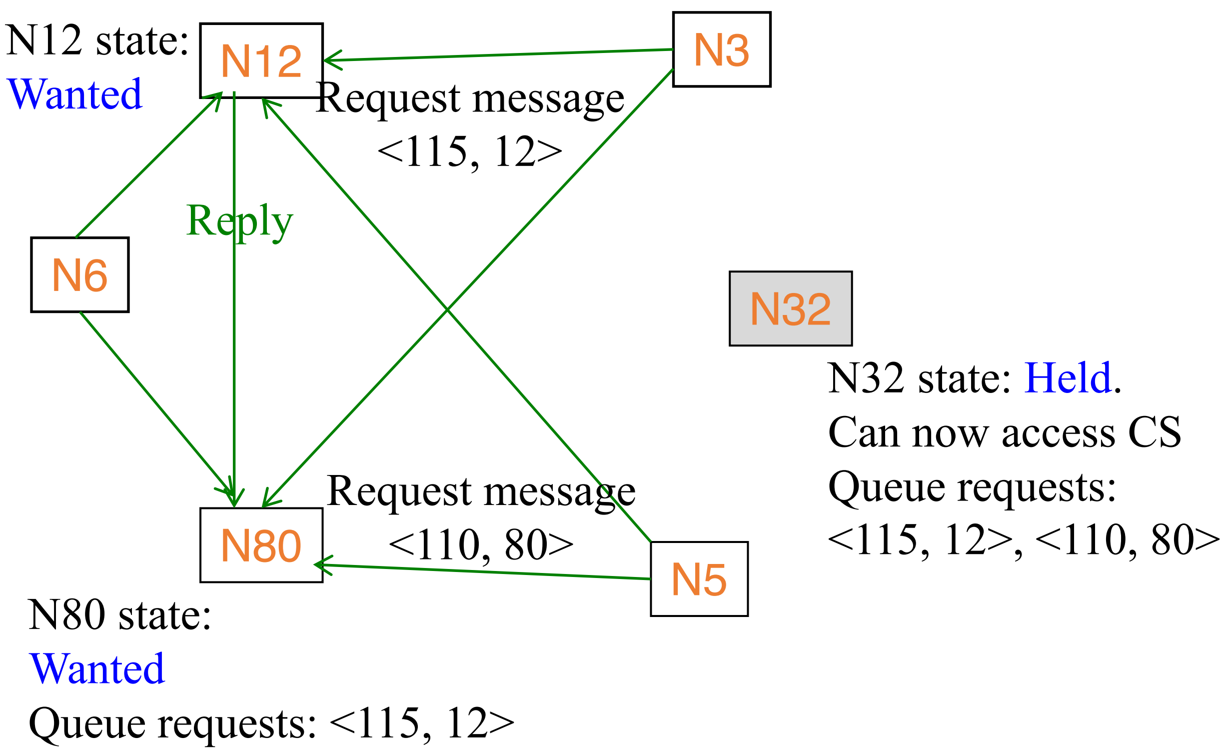
Example: Ricart-Agrawala Algorithm



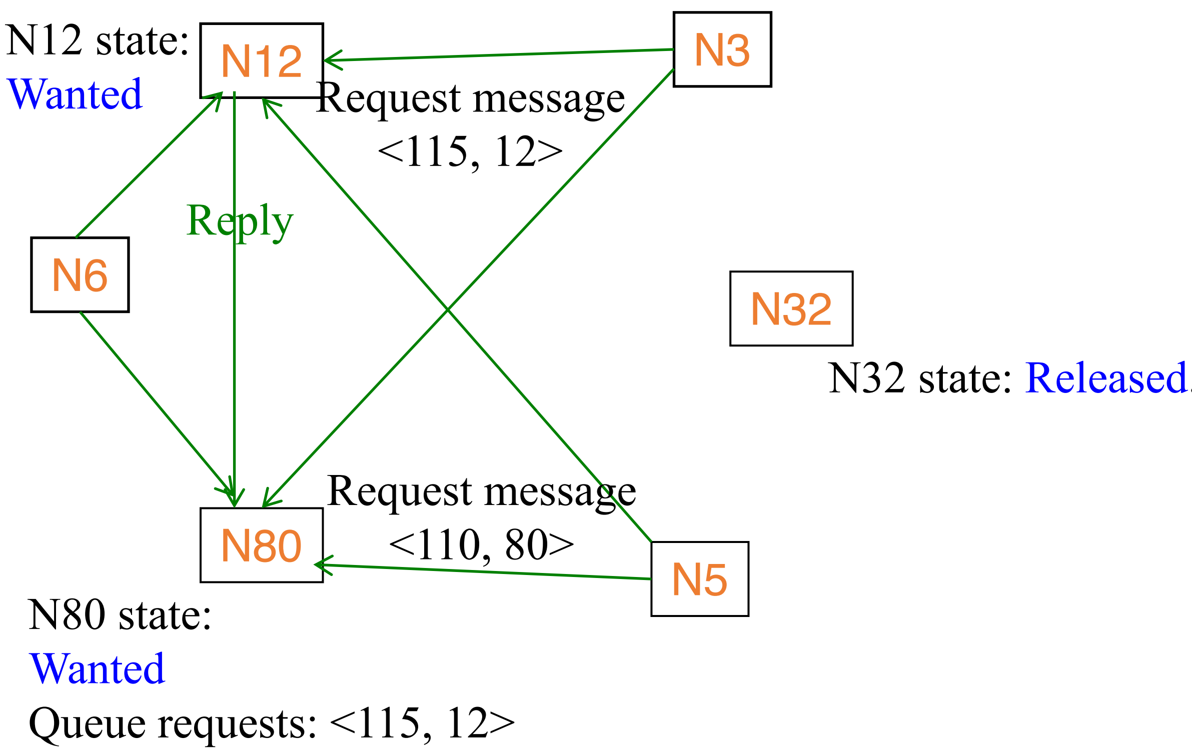
Example: Ricart-Agrawala Algorithm



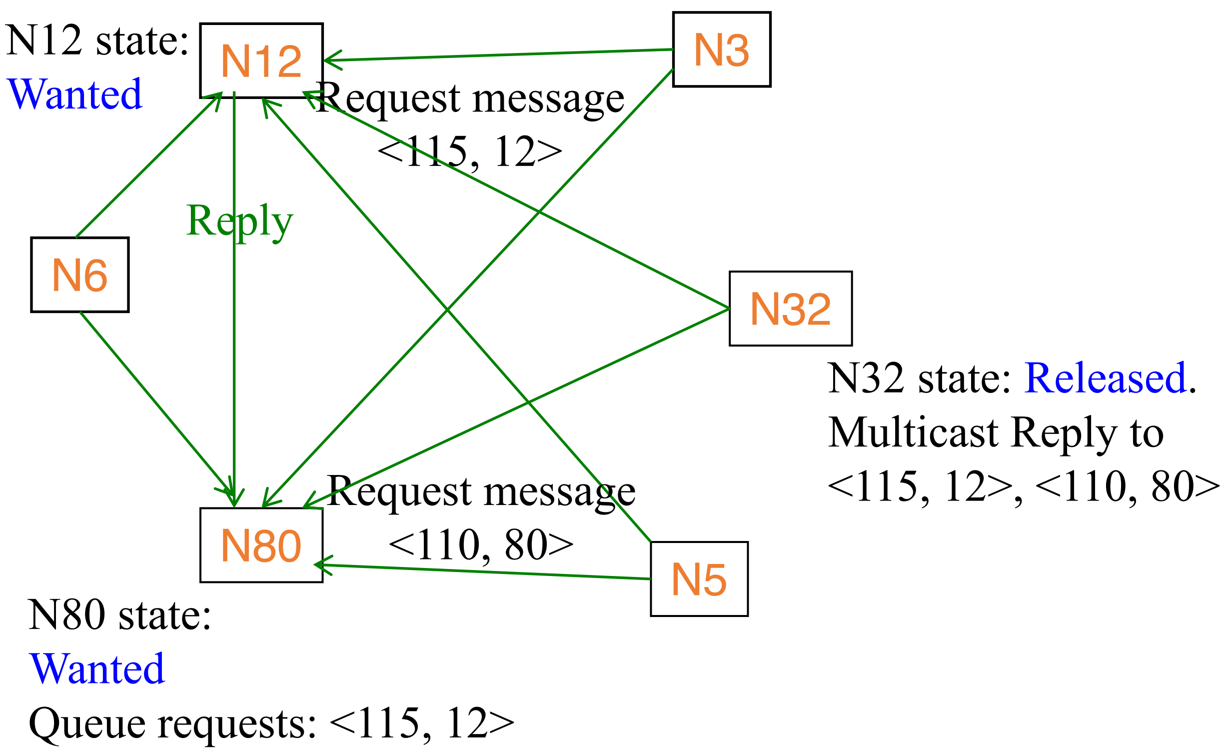
Example: Ricart-Agrawala Algorithm



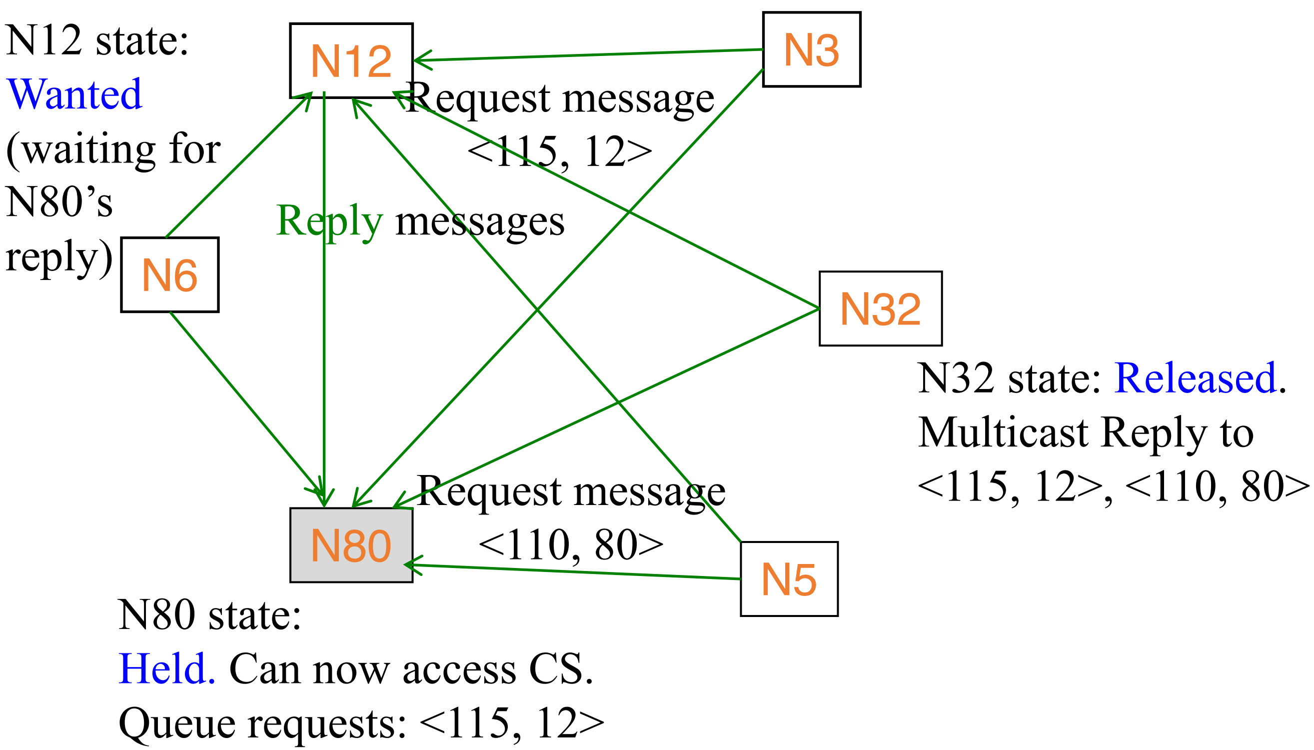
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



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?
      - 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

- Safety
  - Two processes P*i* and P*j* cannot both have access to CS.
- Liveness
  - Worst-case: wait for all other (*N-1*) processes to send Reply.
- Ordering
  - Requests with lower Lamport timestamps are granted earlier.

Analysis: Ricart-Agrawala's Algorithm

- Safety
  - Two processes P*i* and P*j* cannot both have access to CS.
- Liveness
  - Worst-case: wait for all other (*N-1*) processes to send Reply.
- Ordering
  - Requests with lower Lamport timestamps are granted earlier.

Analysis: Ricart-Agrawala's Algorithm

- Bandwidth:
  - 2*(*N*-1) messages per enter operation
    - *N*-1 unicasts for the multicast request + *N*-1 replies
    - Maybe fewer depending on the multicast mechanism.
  - *N*-1 unicasts for the multicast release per exit operation
    - Maybe fewer depending on the multicast mechanism.
- Client delay:
  - one round-trip time
- Synchronization delay:
  - one message transmission time
- *Client and synchronization delays have gone down to O(1).*
- *Bandwidth usage is still high. Can we bring it down further?*

Mutual exclusion in distributed systems

- Our focus today: Classical algorithms for mutual exclusion in distributed systems.
  - Central server algorithm
  - Ring-based algorithm
  - Ricart-Agrawala Algorithm
  - Maekawa Algorithm

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 a time.

Maekawa's Voting Sets

- Each process P_i is associated with a *voting set* V_i (subset of processes).
- Each process belongs to its own voting set.
- *The intersection of any two voting sets must be non-empty.*

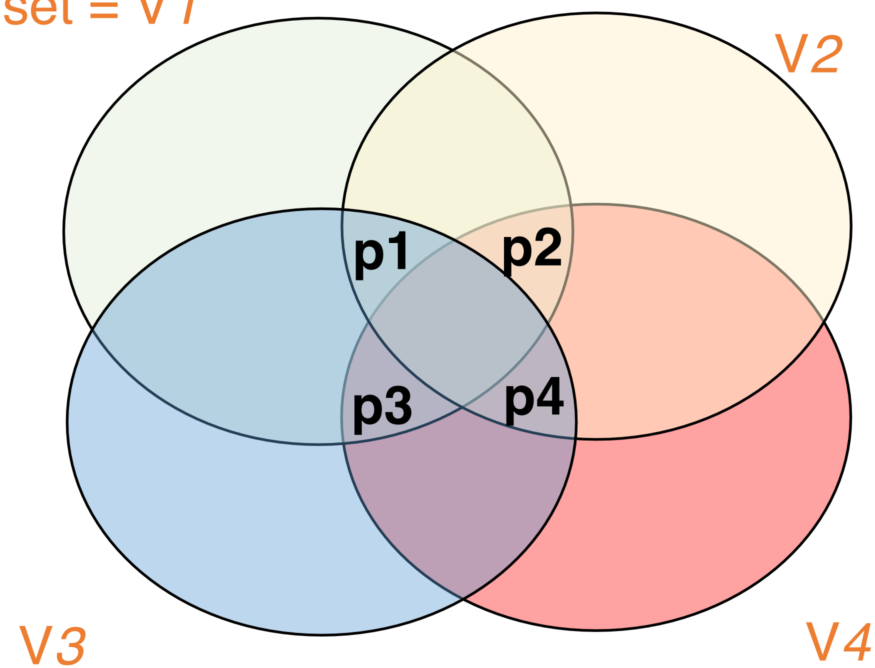
A way to construct voting sets

One way of doing this is to put N processes in a √N by √N matrix and for each Pi, its voting set Vi = row containing Pi + column containing Pi.

Size of voting set = 2*√N-1.

p1	p2
p3	p4

P*1*'s voting set = V*1*



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*:

  Ki

  - 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 (contd.)

- 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

Size of Voting Sets

- Each voting set is of size *K*.
- Each process belongs to *M* other voting sets.
- Maekawa showed that $K = M = approx.\ \sqrt{N}$ works best.

Optional self-study: Why √N ?

- Let each voting set be of size K *and* 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
  - $K*N = M*N \Rightarrow K = M$ (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))
    - $K \sim \sqrt{N}$

Size of Voting Sets

- Each voting set is of size *K*.
- Each process belongs to *M* other voting sets.
- Maekawa showed that $K=M=$*approx.* $\sqrt{N}$ works best.
- Matrix technique gives a voting set size of $2*\sqrt{N}-1 = O(\sqrt{N})$.

Performance: Maekawa Algorithm

- Bandwidth
    - $2K = 2\sqrt{N}$ messages per enter
    - $K = \sqrt{N}$ messages per exit
    - Better than Ricart and Agrawala's ($2*(N-1)$ and $N-1$ messages)
    - $\sqrt{N}$ quite small. $N \sim 1$ million => $\sqrt{N} = 1K$
- Client delay:
    - One round trip time
- Synchronization delay:
    - 2 message transmission times

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

- Does not guarantee liveness, since can have a *deadlock.*
- *System of 6 processes {0,1,2,3,4,5}. 0,1,2 want to enter critical section:*
  - V_0= {0, 1, 2}:
    - 0, 2 send reply to 0, but 1 sends reply to 1;
  - V_1= {1, 3, 5}:
    - 1, 3 send reply to 1, but 5 sends reply to 2;
  - V_2= {2, 4, 5}:
    - 4, 5 send reply to 2, but 2 sends reply to 0;
- Now, 0 waits for 1's reply, 1 waits for 5's reply (5 waits for 2 to send a release), and 2 waits for 0 to send a release. Hence, deadlock!

Analysis: Maekawa Algorithm

- 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*.
- Liveness
  - Not satisfied. Can have deadlock!
- Ordering:
  - Not satisfied.

Breaking deadlocks

- Maekawa algorithm can be extended to break deadlocks.
- Compare Lamport timestamps before replying (like Ricart-Agrawala).
- But is that enough?
  - *System of 6 processes {0,1,2,3,4,5}. 0,1,2 want to enter critical section:*
    - V_0= {0, 1, 2}: 0, 2 send reply to 0, but 1 sends reply to 1;
    - V_1= {1, 3, 5}: 1, 3 send reply to 1, but 5 sends reply to 2;
    - V_2= {2, 4, 5}: 4, 5 send reply to 2, but 2 sends reply to 0;
  - Suppose (L1, P1) < (L0, P0) < (L2, P2).
  - *Deadlock can still happen based on when messages are received.*
    - P5 receives P2's request before P1's, and replies back to P2 first.
- ***We need a way to take back the reply.***

Breaking deadlocks

- To be continued in next class.