

CS 425 / ECE 428

Distributed Systems

Fall 2026

Aishwarya Ganesan & Ram Kesavan

Lecture 19: Mutual Exclusion

Ack: Indranil Gupta

All slides © IG

Announcements

- **Midterm Survey closes 11:59pm Nov 2**
- Reminders:
 - HW3 due 11/1 at 11:59pm (Sunday)
 - **HW3 was updated on 10/26**
 - MP3 due 11/8 at 11:59pm (Sunday)
 - Demos 11/9
 - You should have started on both by now.
- **Final exam 12/15:** 7pm-10pm (on-campus) 4pm-7pm (Chicago).
 - **DRES students:** please book your slot ASAP

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 their Mobile App
 - Both Apps read initial amount of \$1,500 concurrently from the bank's cloud server
 - Both Apps add \$10,000 to this amount (locally in the App)
 - Both write the final amount \$11,500 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 their Mobile App.
 - Both Apps read initial amount of \$1,500 concurrently from the bank's cloud server
 - Both Apps add \$10,000 to this amount (locally in the App)
 - Both write the final amount \$11,500 to the server
 - **You lost \$10,000!**
- The Apps need *mutually exclusive* access to your account entry at the server
 - Or mutually exclusive access to the code execution that modifies the account entry

More Uses of Mutual Exclusion

- **Distributed File systems**
 - Locking of files and directories
- **Accessing objects** in a safe and consistent way
 - Ensure at most one server has access to object at any point of time
- **Server coordination**
 - Work partitioned across servers
 - Servers coordinate using locks
- **In industry**
 - Chubby is Google's locking service
 - Many cloud stacks use Apache Zookeeper for coordination among servers

Problem Statement for Mutual Exclusion

- *Critical Section* Problem: Piece of code (in all processes) that at most one process is executing at any point in time
- Each process calls three functions in sequence:
 - `enter( )` to enter the critical section (CS)
 - `AccessResource( )` to run all the code in the critical section
 - `exit( )` to exit the critical section

Our Bank Example

App1:

```
enter( );  
// AccessResource( )  
obtain bank balance;  
add deposited cheque;  
update bank balance;  
// AccessResource( ) end  
exit( );
```

App2:

```
enter( );  
// AccessResource( )  
obtain bank balance;  
add deposited cheque;  
update bank balance;  
// AccessResource( ) end  
exit( );
```

Solving Mutual Exclusion

Need to guarantee 3 properties:

- **Safety** (essential) – At most one process executes the 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

Simple for single OS/VM/machine:

semaphores, mutexes, condition variables, monitors, etc.

But in a Distributed System

- Cannot share variables like semaphores, mutexes, etc.
- So, how do we support mutual exclusion?
- Processes communicate by passing messages
 - Design mutual exclusion primitives using message passing

System Model

- First, specify our 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
 - Fault-tolerant variants exist in literature.

Central Solution

- Elect a leader
 - Use one of our election algorithms!
- Leader 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:
 - `enter( )`
 - Send a request to leader
 - Wait for token from leader
 - `AccessResource( )` // execute critical section
 - `exit( )`
 - Send back token to leader

Central Solution: Leader Actions

- On receiving a request from process P_i
 - if** (leader has token)
 - Send token to P_i
 - else**
 - Add P_i to queue
- On receiving the token from process P_i
 - if** (queue is not empty)
 - Dequeue waiting process (say P_j)
 - Send token to P_j
 - else**
 - Retain token

Analysis of Central Algorithm

- Safety?
 - At most one process in CS
- Liveness?
 - Every request for CS granted eventually
- Ordering?

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 leader

Analyze Performance

Efficient mutual exclusion algorithms use fewer messages, and make processes wait for shorter durations to access resources. 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 waiting, or in the critical section

We analyze only 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

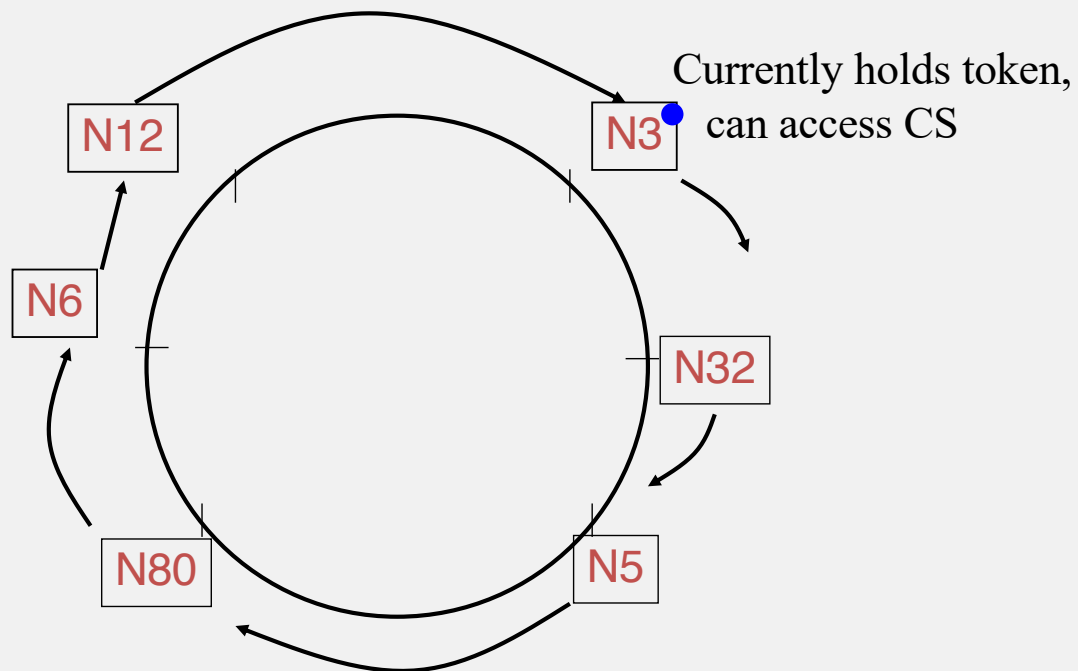
Analysis of Central Algorithm

- **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 at enter operation (when *no* other process is in, or waiting)
 - 1 round trip time (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 leader is...

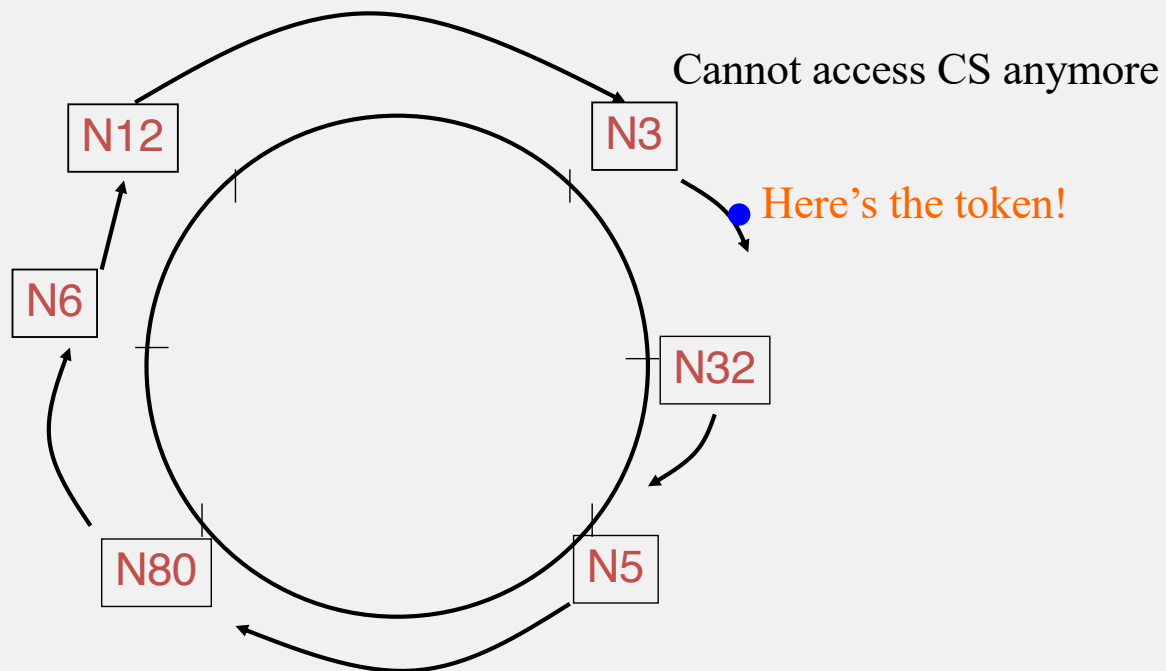
- The performance bottleneck, and
- A SPoF (single point of failure)

Ring-based Mutual Exclusion



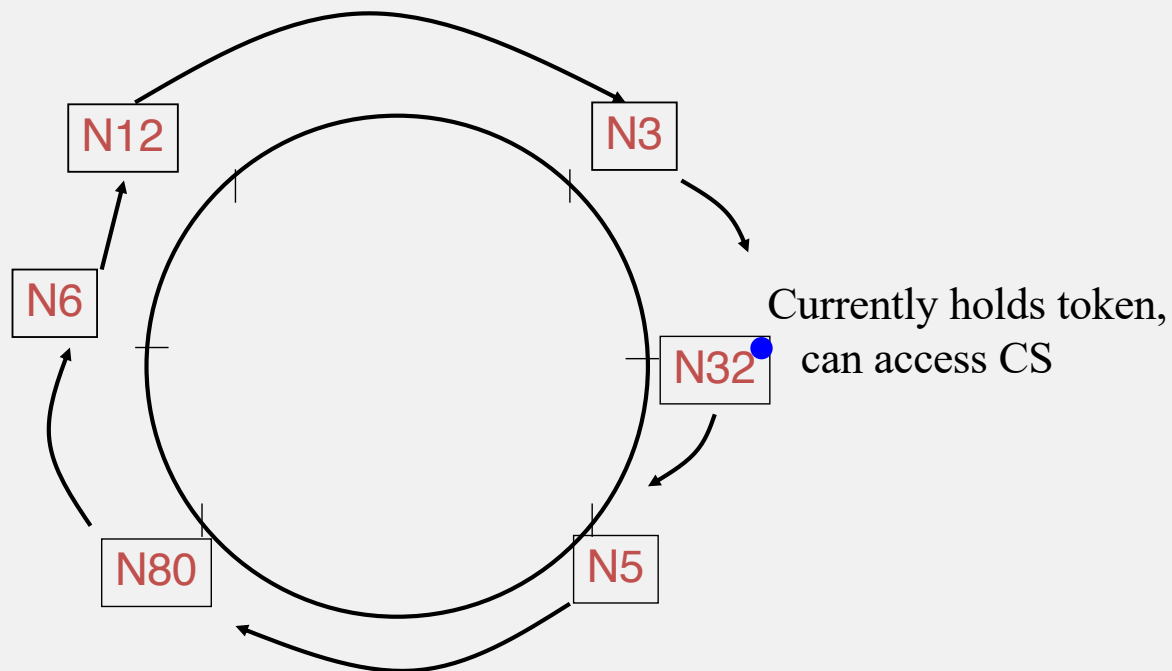
Token: ●

Ring-based Mutual Exclusion



Token: ●

Ring-based Mutual Exclusion



Token: ●

Ring-based Mutual Exclusion

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

Analysis of Ring-based Mutual Exclusion

- Safety: at most one process in CS
- Liveness: every request for CS granted eventually
- Bandwidth: #messages
 - Per enter()
 - Per exit()

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 “grant” message to requesting process
 - 1 “release” message sent per exit()
 - However, wasted bandwidth!

Analysis of Ring-Based Mutual Exclusion (2)

- Client delay: after entering `enter( )`
 - Best case:
 - Worst case:
- Synchronization delay between one process' `exit( )` from the CS and the next process' `enter( )`:
 - Best case:
 - Worst case:

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()`

Next

- Client/Synchronization delay to access CS still $O(N)$ in Ring-Based approach. N can be very large.
- Can we make this faster?

System Model: Reminder

- Each pair of processes is connected by reliable channels (such as TCP)
- Messages from a source to a destination are eventually delivered, and in FIFO order
- Processes do not fail

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

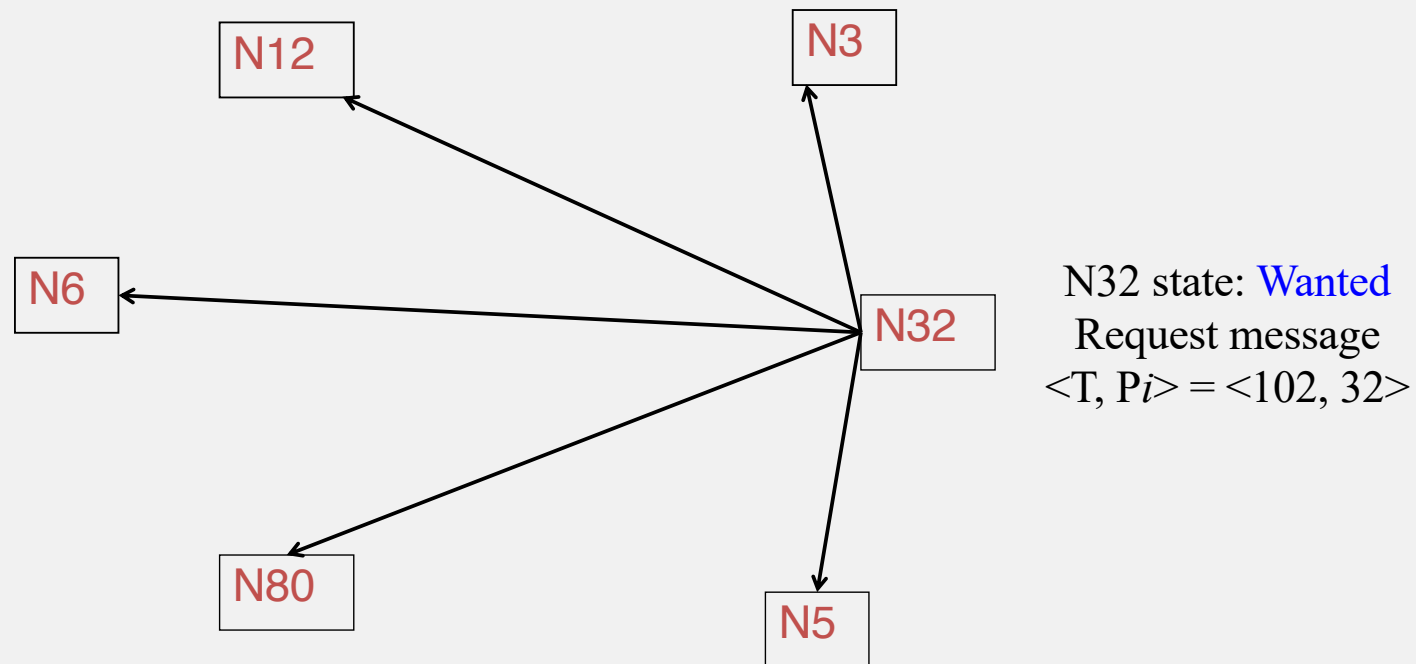
Key Idea: Ricart-Agrawala Algorithm

- enter() at process P_i
 - multicast a request to all processes
 - Request: $\langle T, P_i \rangle$, 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
- $\langle T, P_i \rangle$ is used lexicographically: P_i in request $\langle T, P_i \rangle$ is used to break ties (since Lamport timestamps are not unique for concurrent events)

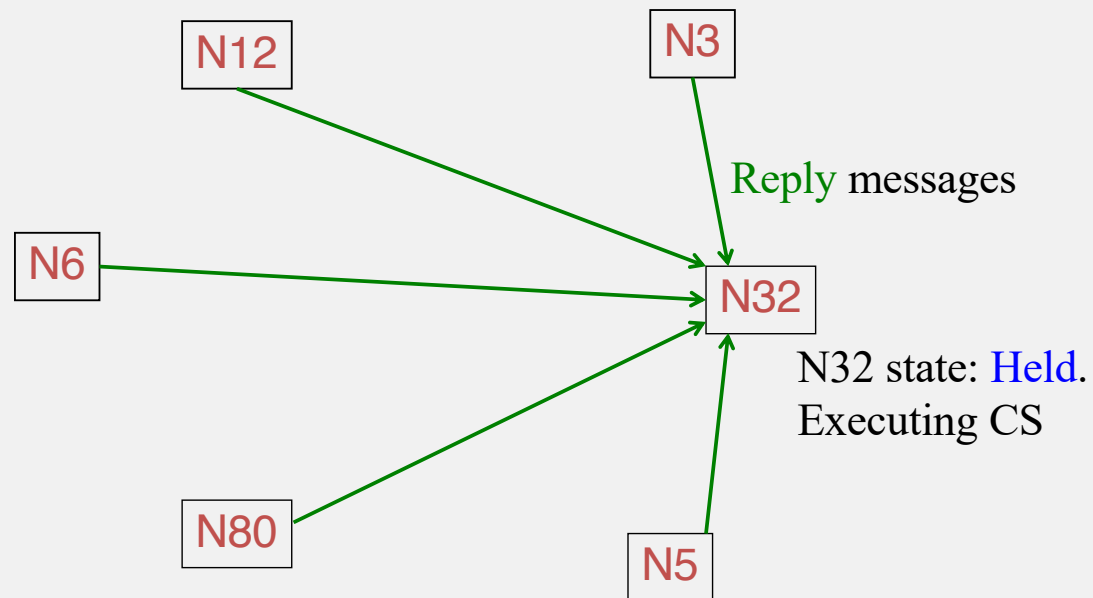
Messages in RA Algorithm

- Each process maintains: state, queue, and Lamport timestamp
- enter() at process P_i
 - state = Wanted
 - multicast Request $\langle T_i, i \rangle$ to all processes, where T_i = current Lamport timestamp at P_i
 - wait until all processes send back “Reply”
 - state = Held and enter the CS
- On receipt of a Request $\langle T_j, j \rangle$ at P_i ($i \neq j$):
 - if (state = Held) or (state = Wanted and $(T_i, i) < (T_j, j)$)
 - // T_i is my pending request's (*not my current*) timestamp
 - add request to local queue (of waiting requests)
 - else send “Reply” to P_j
- exit() at process P_i
 - state = Released and send “Reply” to all queued requests // er, not really a queue! ☺

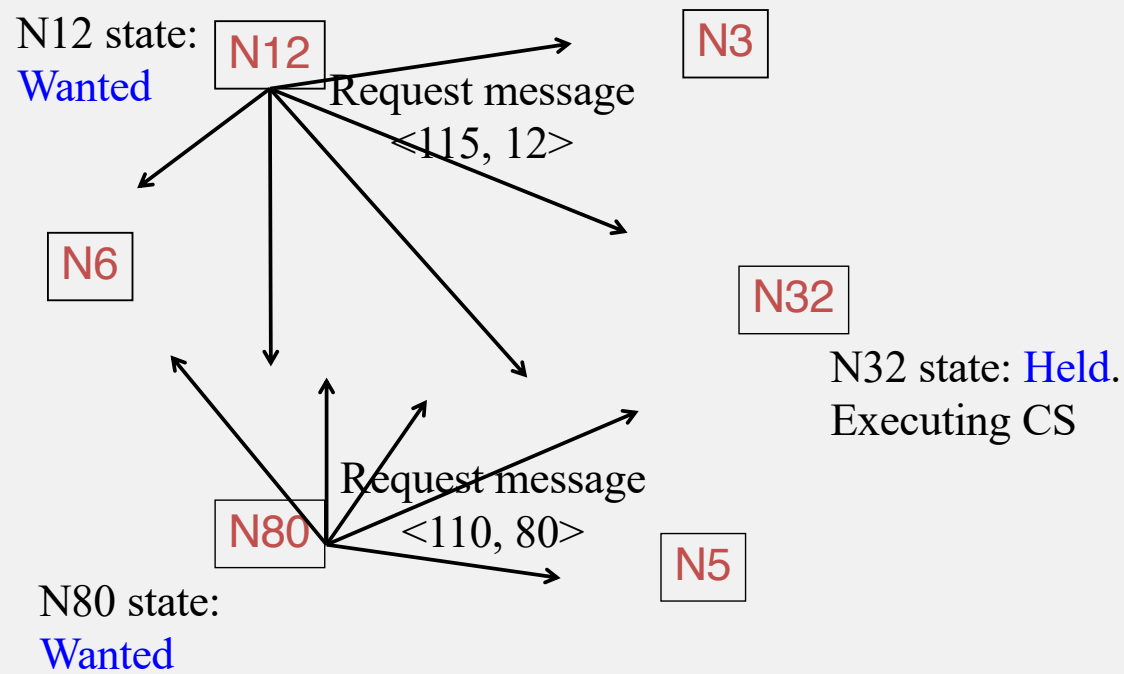
Example: Ricart-Agrawala Algorithm



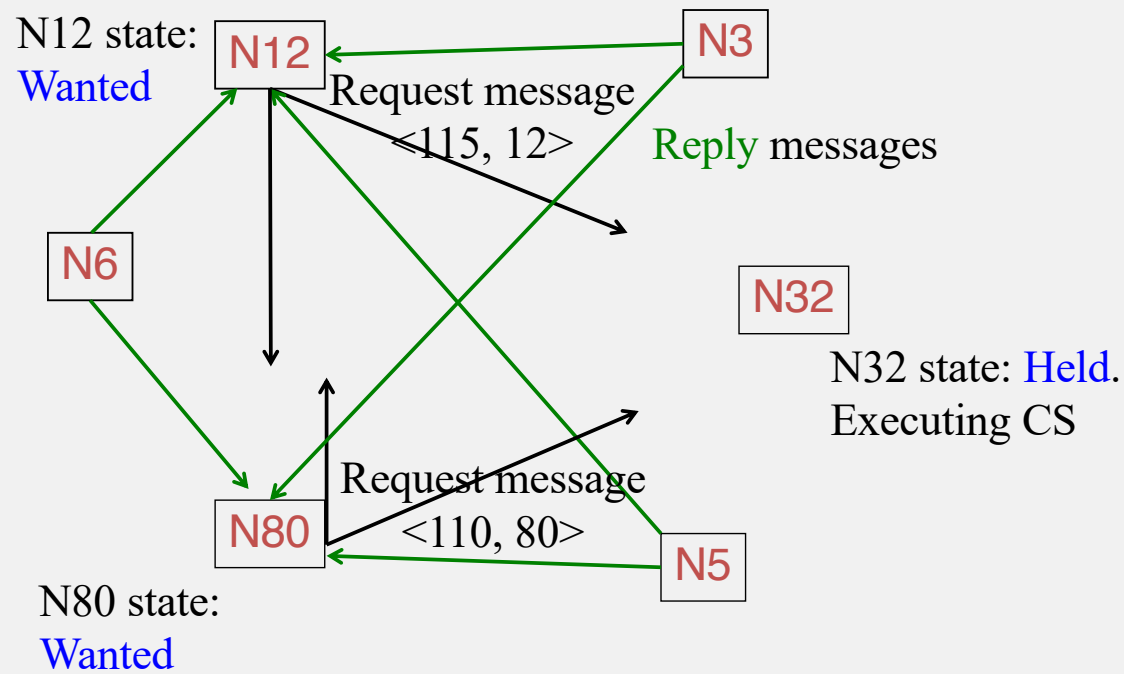
Example: Ricart-Agrawala Algorithm



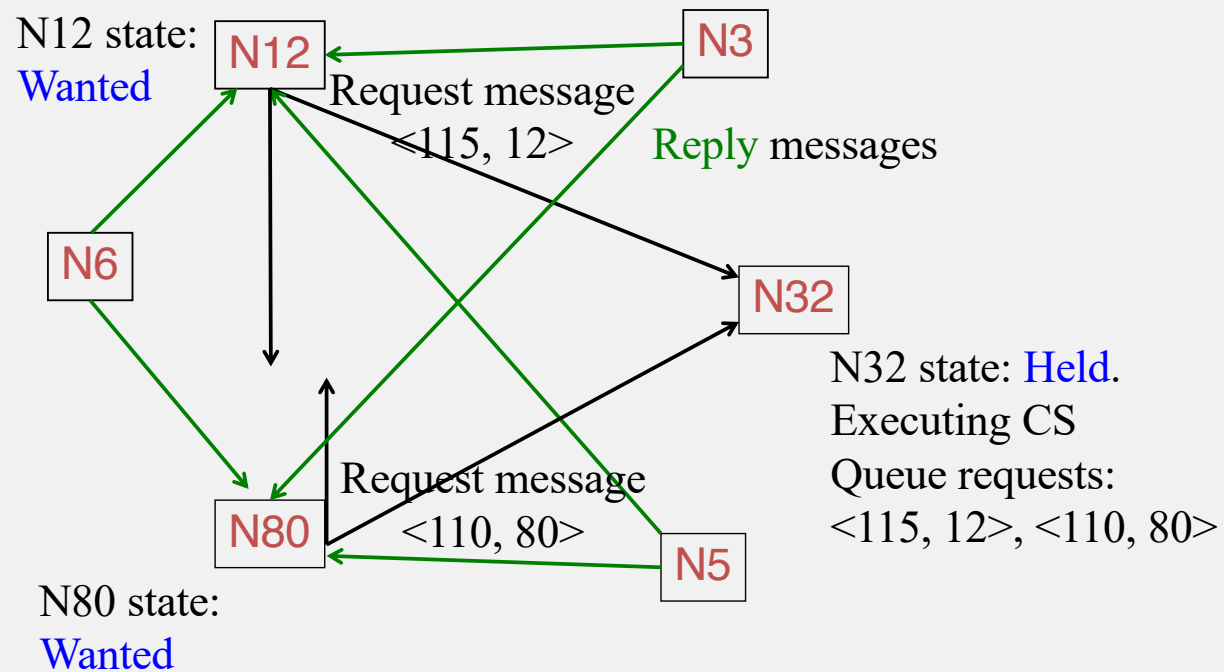
Example: Ricart-Agrawala Algorithm



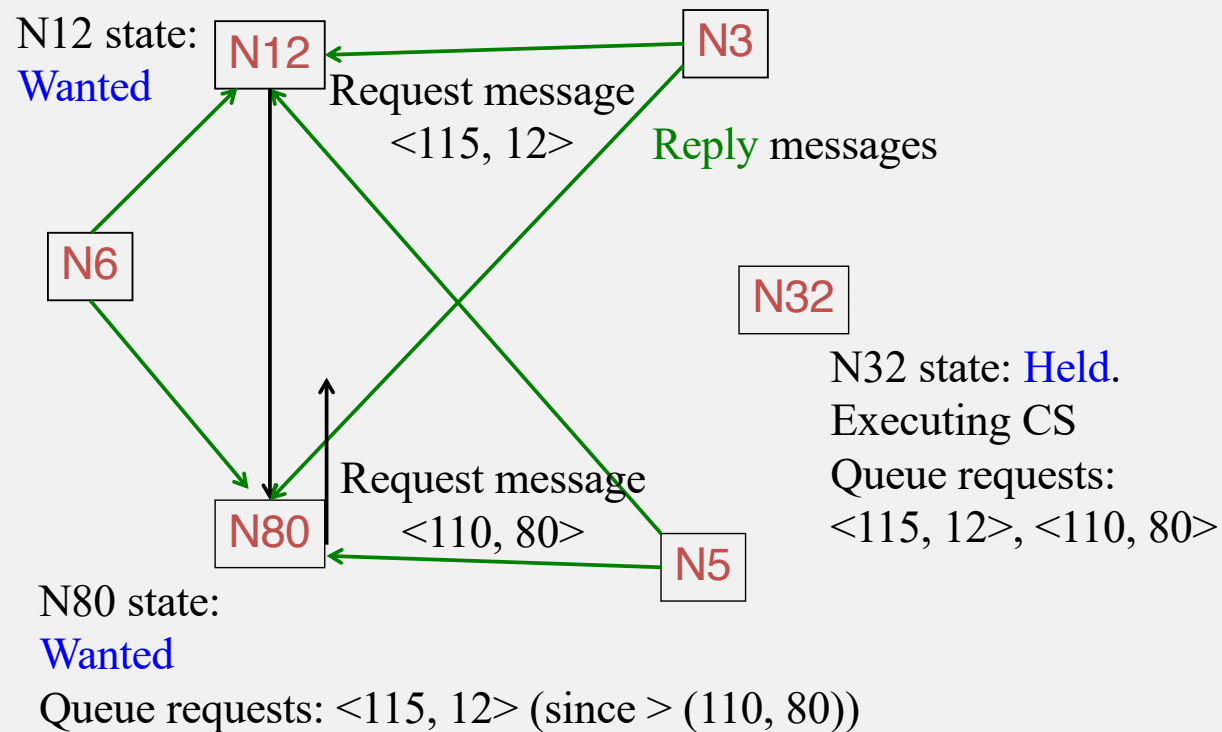
Example: Ricart-Agrawala Algorithm



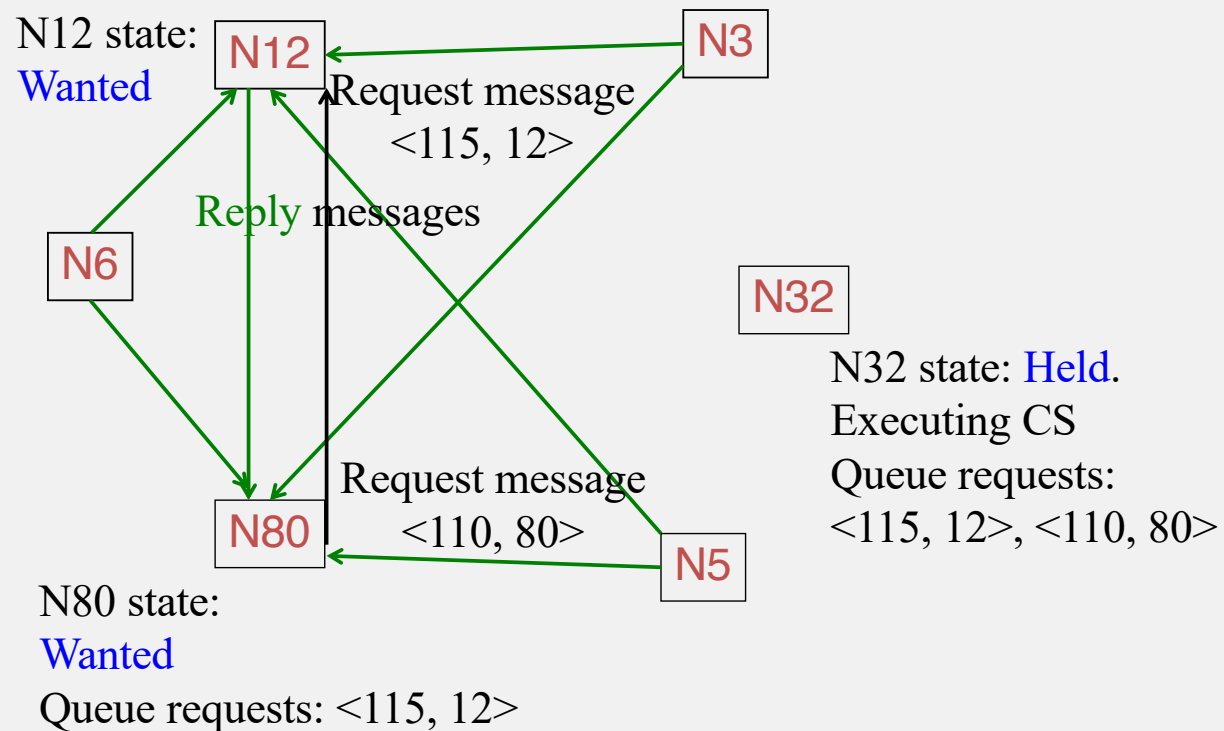
Example: Ricart-Agrawala Algorithm



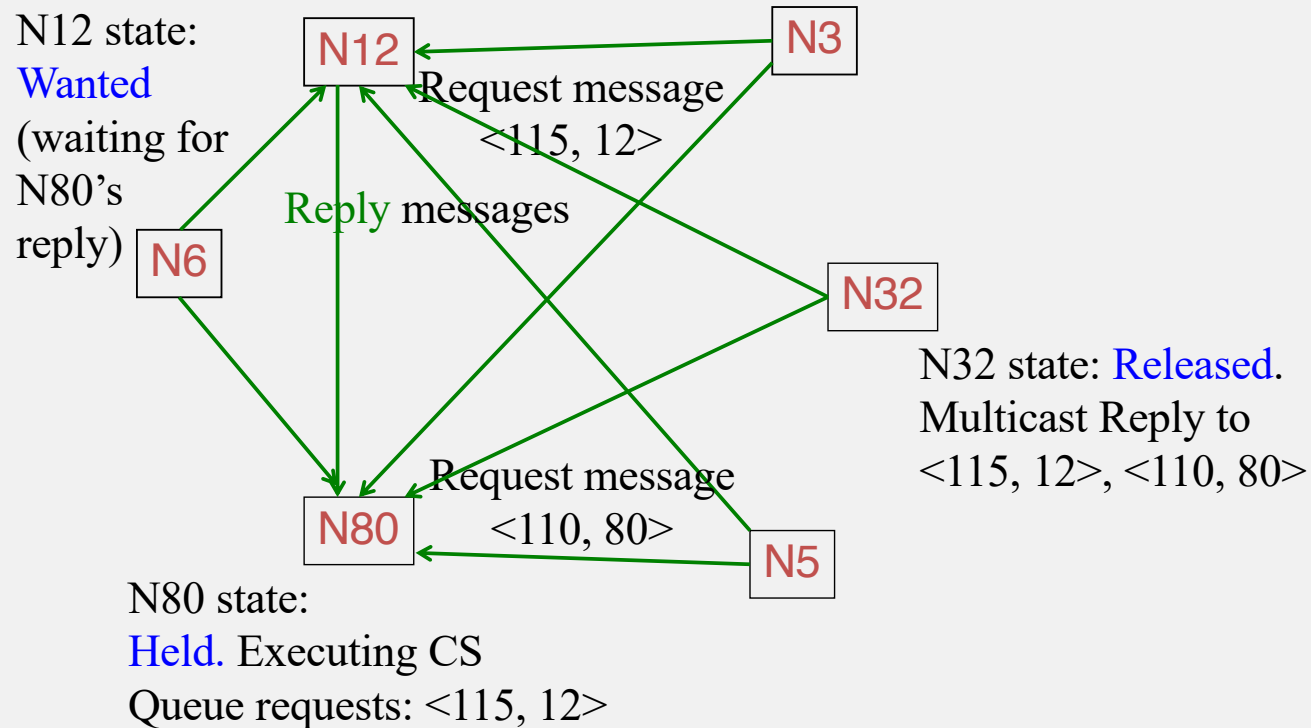
Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



Example: Ricart-Agrawala Algorithm



Analysis: Ricart-Agrawala's Algorithm

- Safety: Two processes 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?
 - Not really. Causality implies that P_i 's Lamport timestamp would have incremented to T_j+1
 - So, this situation cannot arise

Analysis: Ricart-Agrawala's Algorithm (2)

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

Performance: Ricart-Agrawala's Algorithm

- Bandwidth: how many messages per enter() operation?
- Client delay?
 - Delay incurred at enter()
 - No other process interested in CS
- Synchronization delay?
 - Delay between one process' exit() from the CS and the unblocking of another process' enter()
 - No other process interested in CS

Performance: Ricart-Agrawala's Algorithm

- Bandwidth: $2*(N-1)$ 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)
 - Exit() Operation: $N-1$ unicast messages per
 - 1 multicast if the underlying network supports multicast
- Client delay: one round-trip time (parallel)
- Synchronization delay: one message transmission time

Ok, but ...

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

Maekawa's Algorithm: Key Idea

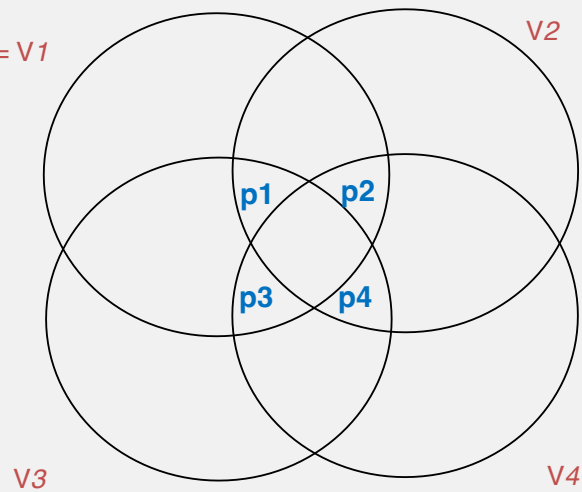
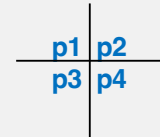
- Ricart-Agrawala requires replies from *all* processes in group
- Instead, get replies from only *some* processes in group
- But ensure that only process one is given access to CS (Critical Section) at a time

Maekawa's Voting Sets

- Each process P_i is associated with a voting set 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 voting sets
- Maekawa showed that optimally $K=M$ and $K \sim \sqrt{N}$
- One way: 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

P1's voting set = V1



Maekawa: Key Differences From Ricart-Agrawala

- Each process requests permission from only its voting set members
 - Not from all
- Each process can give its permission to at most one process at any point in time
 - Not to all
 - In other words, each process has exactly 1 vote to give

Actions

- Each process maintains: state, queue, voted (boolean)
- 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 (vote)** to all processes in V_i

Actions (2)

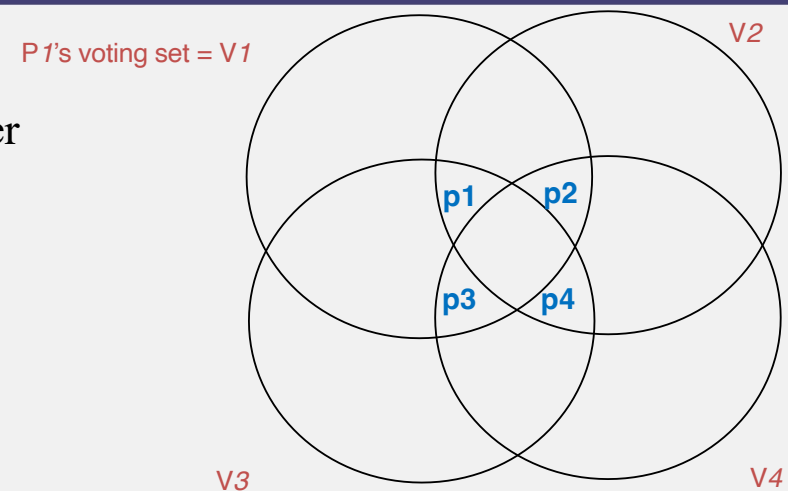
- When P_i receives a Request from P_j :
 - if** (state == Held OR voted = true)
 - enqueue Request
 - else**
 - send **Reply** to P_j
 - voted = true
- When P_i receives a Release from P_j :
 - if** (queue empty)
 - voted = false
 - else**
 - dequeue head of queue, say P_k
 - send **Reply** *only* to 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

- Bandwidth
 - $2\sqrt{N}$ 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 $\Rightarrow \sqrt{N} = 1K$
- 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 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))
 - $K \sim \sqrt{N}$

Failures?

- There are fault-tolerant versions of the algorithms we've discussed
 - E.g., Maekawa
- One other way to handle failures: Use Paxos-like approaches!

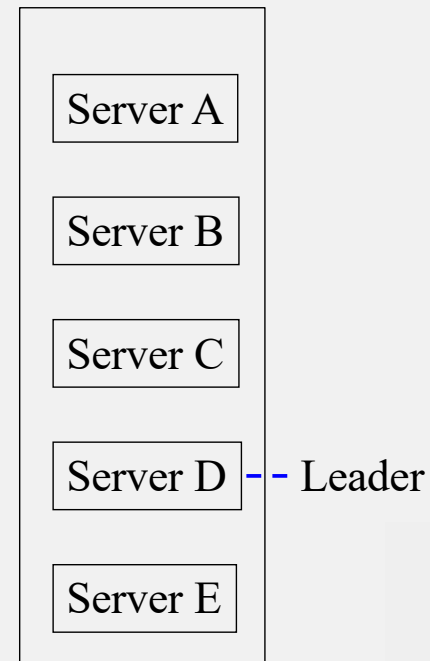
Chubby

- Google's system for locking
- Used underneath Google's systems like BigTable, Megastore, etc.
- Not open-sourced but published
 - Apache Zookeeper was inspired by this paper
- Chubby provides *Advisory* locks only
 - Doesn't guarantee mutual exclusion unless every client checks lock before accessing resource

Reference: <http://research.google.com/archive/chubby.html>

Chubby (2)

- Used not only for locking but for also writing small configuration files
- Paxos-like consensus protocol to elect a Leader
- Same info replicated in all servers
- Clients send read requests to Leader, which serves it locally
 - Aggressive client-side caching + invalidations
- Clients send write requests to Leader, which replicates to all servers, gets majority (quorum) among servers, and then acks client
- Leader failure → election protocol is run
- Replica failure → replace it and have it catch up



Summary

- Mutual exclusion important problem in cloud computing systems
- Classical algorithms
 - Central
 - Ring-based
 - Ricart-Agrawala
 - Maekawa
- Industry systems coordination service
 - Proprietary: Chubby
 - Open source: Apache Zookeeper

Announcements

- **Midterm Survey closes 11:59pm Nov 2**
 - **Incentive announced on Piazza if participation $\geq 70\%$**
 - Must login with Illinois email and must have Google Apps @ Illinois enabled.
- **Reminders:**
 - HW3 due 11/1 at 11:59pm (Sunday)
 - MP3 due 11/8 at 11:59pm (Sunday)
 - Demos 11/9
- **Final exam 12/15: 7pm-10pm (on-campus) 4pm-7pm (Chicago).**
 - Please plan accordingly
 - **DRES students:** please book your slot ASAP