Mutual Exclusion in a Distributed Environment

- Mutual exclusion
  - Centralized algorithms
    - Central physical clock
    - Central coordinator
  - Distributed algorithms
    - Time-based event ordering
      - Lamport's algorithm (logical clocks)
      - Ricart & Agrawala's algorithm (" ")
      - Suzuki & Kasimi's algorithm (broadcast)
    - Token passing
      - Le Lann's token-ring algorithm (logical ring)
      - Raymond's tree algorithm (logical tree)
    - Sharing K identical resources
      - Raymond's extension to Ricart & Agrawala's time-based algorithm
  - Atomic transactions (later in course)
- Related — self-stabilizing algorithms, election, agreement, deadlock

Mutual Exclusion in a Distributed Environment — General Requirements

- N processes share a single resource, and require mutually-exclusive access
- Conditions to satisfy:
  - A process holding the resource must release it before it can be granted to another process
  - Requests for the resource must be granted in the order in which they're made
  - If every process granted the resource eventually releases it, then every request will be eventually granted
- Assumptions made:
  - Messages between two processes are received in the order they are sent
  - Every message is eventually received
  - Each process can send a message to any other process

Central Physical Clock

- Provide a single central physical clock, just like in a centralized system
  - Processes request physical timestamps from this clock and use them to order events

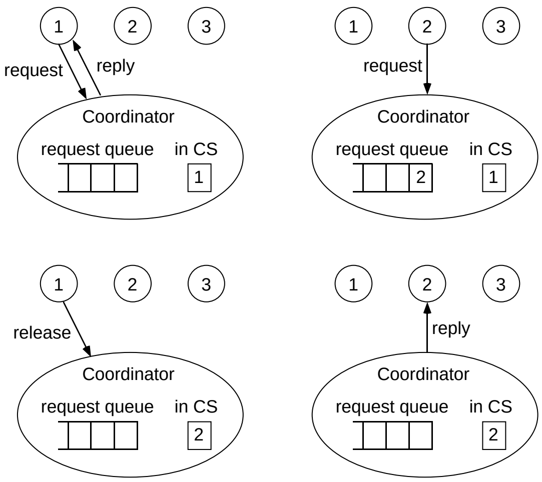
✔ Advantages:
  - Simplicity

✘ Disadvantages:
  - Clock must always be available to provide the requested timestamps
  - Transmission errors can prevent the proper ordering from taking place
  - An accurate estimation of transmission delays is required
  - The degree of accuracy may not be as high as desired

Central Coordinator

- To enter the critical section, a thread sends a *request* message to the central coordinator, and waits for a reply
- When the coordinator receives a request:
  - If **no** other thread is in the critical section, it sends back a *reply* message
  - If another thread **is** in the critical section, the coordinator adds the request to the tail of its queue, and does not respond
- When the requesting thread receives the *reply* message from the coordinator, it enters the critical section
  - When it leaves the critical section, it sends a *release* message to coordinator
  - When the coordinator receives a *release* message, it removes the request from the head of the queue, and sends a *reply* message to that thread

Central Coordinator (cont.)



- Evaluation:
  - 3 messages required to enter CS
    - release, request, reply
  - ✘ Coordinator is a performance bottleneck
  - ✘ Coordinator is a single point of failure
  - ✘ Delay is unconstrained

Lamport's Algorithm (1978)

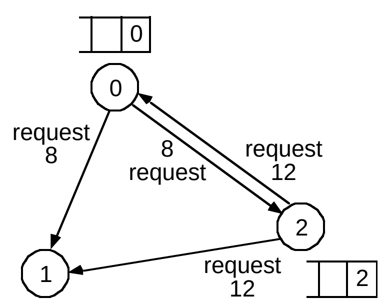
- Each process maintains a request queue, ordered by timestamp value
- Requesting the critical section (CS):
  - When a thread wants to enter the CS, it:
    - Adds the request to its own request queue
    - Sends a timestamped *request* message to all threads in that CS's request set
  - When a thread receives a *request* message, it:
    - Adds the request to its own request queue
    - Returns a timestamped *reply* message
- Executing the CS:
  - A thread enters the CS when **both**:
    - Its own request is at the top of its own request queue (its request is earliest)
    - It has received a *reply* message with a timestamp larger than its request from all other threads in the request set

Lamport's Algorithm (cont.)

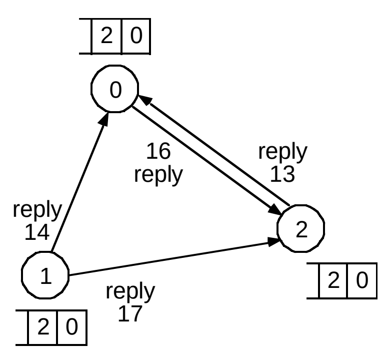
- Releasing the CS:
  - When a thread leaves the CS, it:
    - Removes its own (satisfied) request from the top of its own request queue
    - Sends a timestamped *release* message to all threads in the request set
  - When a thread receives a *release* message, it:
    - Removes the (satisfied) request from its own request queue
    - (Perhaps raising its own message to the top of the queue, enabling it to finally enter the CS)
- Evaluation:
  - 3(N–1) messages required to enter CS
    - (N–1) release, (N–1) request, (N–1) reply
  - ✘ Later…

Lamport's Algorithm (cont.)

- Both threads 0 and 2 request the CS:

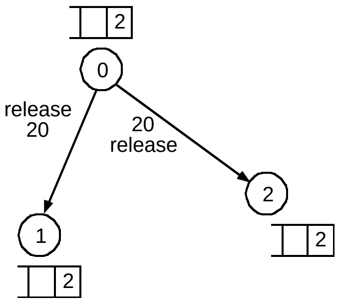


- Everyone replies, thread 0 enters the CS since its request was first:



Lamport’s Algorithm (cont.)

- Thread 0 releases the CS, thread 2 enters it:



Ricart and Agrawala’s Algorithm (1981)

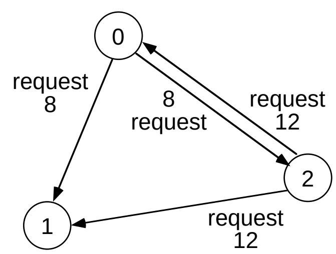
- Requesting the critical section (CS):
  - When a thread wants to enter the CS, it:
    - Sends a timestamped *request* message to all threads in that CS’s request set
  - When a thread receives a *request* message:
    - If it is neither requesting nor executing the CS, it returns a *reply* message
    - If it is requesting the CS, but the timestamp on the incoming request is smaller than the timestamp on its own request, it returns a *reply* message
      - Means the other thread requested first
    - Otherwise, it defers answering the request
- Executing the CS:
  - A thread enters the CS when:
    - It has received a *reply* message from all other threads in the request set

Ricart and Agrawala’s Algorithm (cont.)

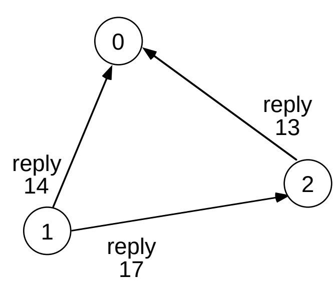
- Releasing the CS:
  - When a thread leaves the CS, it:
    - Sends a *reply* message to all the deferred requests
    - (Thread with next earliest request will now received its last *reply* message and enter the CS)
- Evaluation:
  - 2(N–1) messages required to enter CS
    - (N–1) reply, (N–1) request
- Evaluation (Lamport, Ricart & Agawala):
  - ✘ Distributed performance bottleneck
  - ✘ Now N points of failure
    - If a thread crashes, it fails to reply, which is interpreted as a denial of permission to enter the CS, so everyone waits…
  - ✘ Need up-to-date group communication

Ricart and Agrawala’s Algorithm (cont.)

- Both threads 0 and 2 request the CS:



- Threads 1 and 2 reply, thread 0 defers and enters the CS since its request was first:



- After leaving the CS, thread 0 replies to thread 2, which enters the CS

Raymond's Extension For Sharing K Identical Resources (1987)

- K identical resources, which must be shared among N processes
- Raymond's extension to Ricart and Agrawala's algorithm:
  - A process can enter the CS as soon as it has received N–K *reply* messages
  - Algorithm is generally the same as R&A, with one difference:
    - R&A — *reply* messages arrive only when process is waiting to enter CS
    - Raymond —
      - N–K *reply* messages arrive when process is waiting to enter CS
      - Remaining K–1 *reply* messages can arrive when process is in the CS, after it leaves the CS, or when it's waiting to enter the CS again
      - Must keep a count of number of outstanding *reply* messages, and not count those toward next set of replies

Election Algorithms

- In a distributed system, many algorithms require a permanent or temporary leader:
  - Distributed mutual exclusion:
    - Central coordinator algorithm requires a coordinator
    - Token-ring algorithm, Suzuki-Kasami's broadcast algorithm, and Raymond's tree algorithm require an initial token holder
  - Distributed deadlock detection — maintainer of a global wait-for graph
- If leader fails, must *elect* a new leader
  - Election algorithms assume there is a unique priority number for each thread
  - Goal: elect the highest-priority thread as the leader, tell all active threads
  - Second goal: allow a recovered leader to re-establish control (or at least, to identify the current leader)

Garcia-Molina's Bully Algorithm (1993)

- 3 types of messages:
  - *Election* —announce an election
  - *Answer* — acknowledge election msg.
  - *Coordinator* — announce new coordinator
- The election:
  - A thread begins an election when it notices the coordinator has failed
    - To do so, it sends *election* messages to all threads with a higher priority
  - It then awaits an *answer* message (from a live thread with a higher priority)
    - If none arrives within a certain time, it declares itself the coordinator, and sends a *coordinator* message to all threads with a lower priority
    - If an *answer* message does arrive, it waits a certain time for a *coordinator* message to arrive from the new coordinator
      - If none arrives, it begins another election

Garcia-Molina's Bully Algorithm (cont.)

- Result of the election:
  - If a thread receives a *coordinator* message, it accepts the new coordinator
- Participating in an election:
  - If a thread receives an *election* message:
    - It sends back an answer message
    - It begins another election (with its higher-ups) unless it has already begun one
- Failed threads:
  - When one restarts, it begins an election
    - Unless it knows it has the highest priority, in which case it just sends out *coordinator* messages to re-establish control
- Evaluation:
  - N–2 messages in best case
  - $O(N^2)$ messages in worst case

Garcia-Molina's Bully Algorithm (cont.)