CS 425/ECE 428/CSE424
Distributed Systems
(Fall 2009)

Lecture 7

Distributed Mutual Exclusion

Section 12.2

Klara Nahrstedt

Acknowledgement

- **The slides during this semester are based on ideas and material from the following sources:**
  - **Slides prepared by Professors M. Harandi, J. Hou, I. Gupta, N. Vaidya, Y-Ch. Hu, S. Mitra.**
  - **Slides from Professor S. Gosh's course at University o Iowa.**

Administrative

- **Homework 1 posted September 3, Thursday**
  - **Deadline, September 17 (Thursday)**
- **MP1 posted September 8, Tuesday**
  - **Deadline, September 25 (Friday), 4-6pm Demonstrations**

Plan for today

- **Distributed Mutual Exclusion**
  - **Centralized coordinator-based approach**
  - **Token-based approach**
  - **Ricart and Agrawala's timestamp algorithm**
  - **Maekawa's algorithm**
  - **Raymond's algorithm**

Distributed Mutual Exclusion: Performance Evaluation Criteria

- ***Bandwidth***: **the total number of messages sent in each *entry* and *exit* operation.**
- ***Delay:***
  - ***Client delay***: **the delay incurred by a process at each entry and exit operation (when *no* other process is waiting)**
  - ***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)**
- **These translate into *throughput* -- the rate at which the processes can access the critical section, i.e., *x* processes per second.**

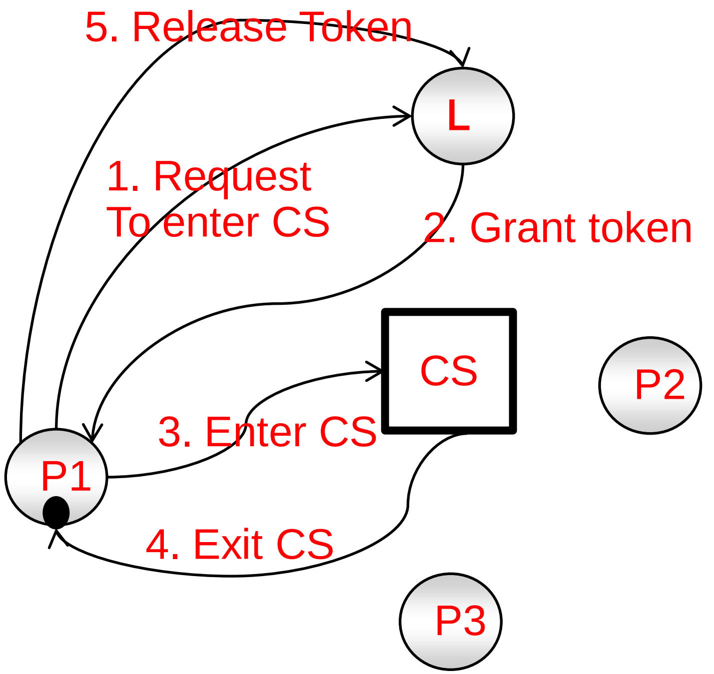
(these definitions more correct than the ones in the textbook)

Assumptions

- For all the algorithms studied, we assume
  - Reliable FIFO channels between every process pair
    - Each pair of processes is connected by reliable channels (such as TCP). Messages are eventually delivered to recipients' input buffer in FIFO order.
  - Processes do not fail.

Centralized Control of Distributed Mutual Exclusion

- **A central coordinator or leader L**



  - **Is appointed or elected**
  - **Grants permission to enter CS & keeps a queue of requests to enter the CS.**
  - **Ensures only one process at a time can access the CS**
  - **Separate handling of different CS's**

Centralized control of Distributed ME

- **Operations (*token* gives access to CS)**
  - **To enter a CS**

    Send a request to L & wait for <u>token</u>.
  - **On exiting the CS**

    Send a message to the coord to release the token.
  - **Upon receipt of a request,**

    if no other process has the token, L replies with the token; otherwise, L queues the request.
  - **Upon receipt of a release message**

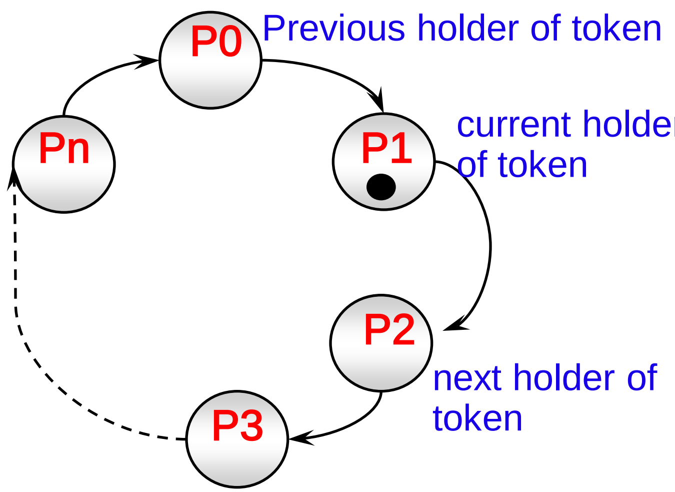
    L removes the oldest entry in the queue (if any) and replies with a token.

- **Features:**
  - **Safety, liveness and order are guaranteed**
  - **It takes 3 messages per entry + exit operation.**
  - **Client delay: one round trip time (request + grant)**
  - **Synchronization delay: one round trip time (release + grant)**
  - **The coordinator becomes performance bottleneck and single point of failure.**

Token Ring Approach

- **Processes are organized in a logical unidirectional ring: p_i has a communication channel to $p_{(i+1) mod (n+1)}$.**
- **Operations:**
  - **Only the process holding the token can enter the CS.**
  - **To enter the critical section, wait for the token.**
  - **To exit the CS, p_i sends the token onto its neighbor.**
  - **If a process does not want to enter the CS when it receives the token, it forwards the token to the next neighbor.**

- **Features:**
  - **Safety & liveness are guaranteed, but ordering is not.**
  - **Bandwidth: 1 message per exit**
  - **Client delay: 0 to (N+1) message transmissions.**
  - **Synchronization delay between one process's exit from the CS and the next process's entry is between 1 and N message transmissions.**



Timestamp Approach: Ricart & Agrawala

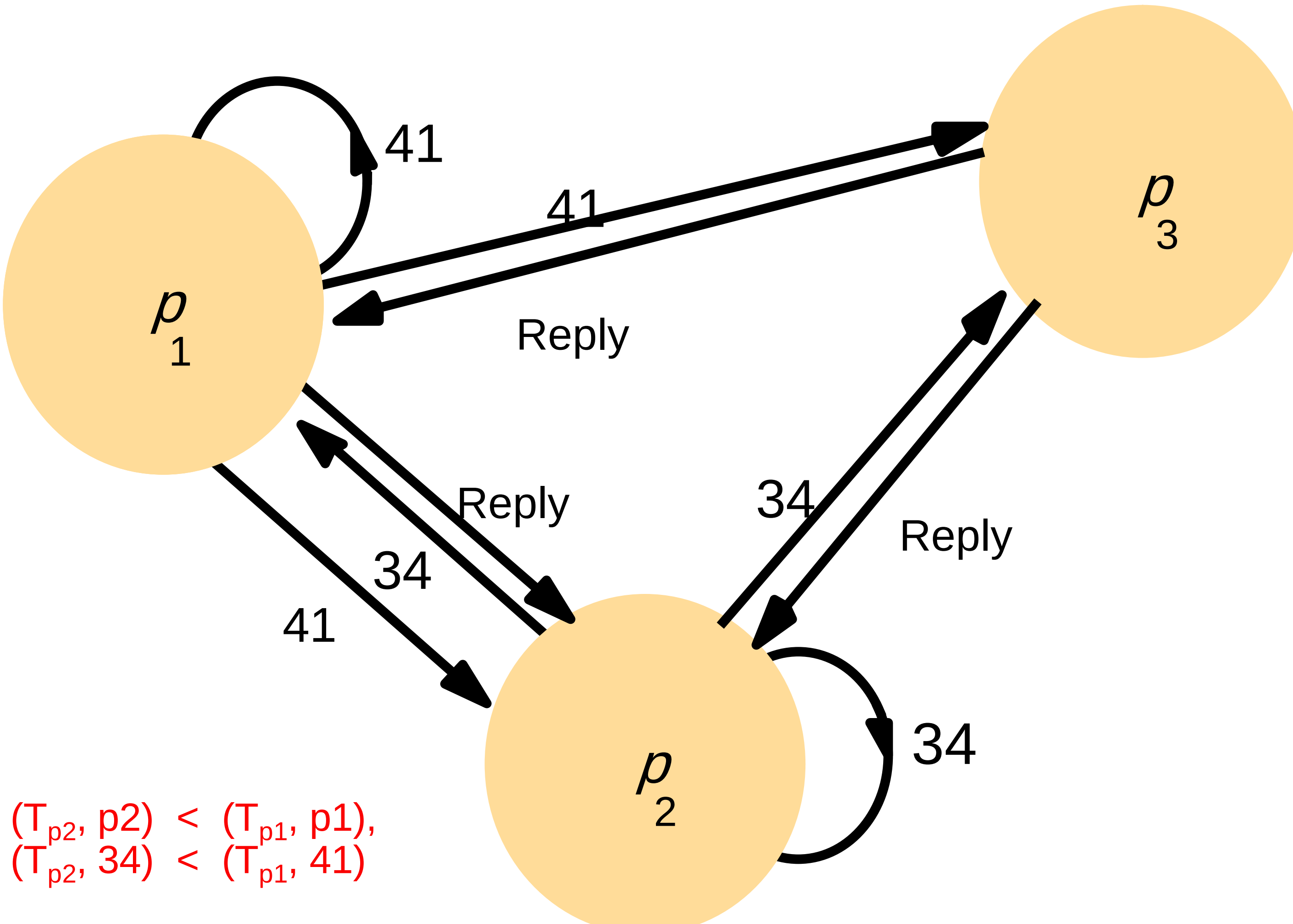
- **Processes requiring entry to critical section multicast a request, and can enter it only when *all* other processes replied positively.**
  - **Messages requesting entry are of the form $<T,p_i>$, where T is the sender's timestamp (from a *Lamport* clock) and p_i the sender's identity (used to break ties in T).**
  - ***state* of a process can be wanted, held, released**
- **p_i to enter the CS**
  - **set state to wanted**
  - **multicast "request" to all processes (include timestamp).**
  - **wait until all processes send back "reply"**
  - **change state to held and enter the CS**
- **On receipt of a request $<T_i, p_i>$ at p_j:**
  - **if (state = held) or (state = wanted & $(T_j, p_j)<(T_i,p_i)$), enqueue request**
  - **else "reply" to p_i**
- **p_i on exiting the CS**
  - **change state to release and "reply" to *all* queued requests.**

Ricart and Agrawala's algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = $(N - 1)$);
    state := HELD;

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Ricart and Agrawala's algorithm



$(T_{p2}, p2) < (T_{p1}, p1)$,
$(T_{p2}, 34) < (T_{p1}, 41)$

Hence enqueue request
From p1

Timestamp Approach: Ricart & Agrawala

- ❖ **Features:**
  - ❖ Safety, liveness, and ordering (causal) are guaranteed (why?)
  - ❖ Messages per entry operation
    - ❖ 2(N-1) = (N-1) unicasts for the multicast request + (N-1) replies;
    - » N messages if the underlying network supports multicast,
  - – Messages per exist operation
    - » N-1 unicast
    - » 1 multicast if the underlying network supports multicast
  - ❖ Client delay:
    - ❖ one round-trip time
  - ❖ Synchronization delay:
    - ❖ one message transmission time.

Timestamp Approach: Maekawa's Algorithm

- **Multicasts messages to a (voting) subset of nodes**
  - **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 is not empty**
  - **Each voting set is of size K**
  - **Each process belongs to M other voting sets**
  - **To access at resource, p_i requests permission from all other processes in its own voting set v_i**

  - **Guarantees safety, not liveness (may deadlock)**
  - **Maekawa showed that K=M= O(√N) works best**

  One way of doing this is to put nodes in a √N by √N matrix and take the union of row & column containing p_i as its voting set.

  Example: N= 16, if we put 16 processes into 4x4 matrix, the voting set for P7 is V_i (P7) = {P3, P5, P6, P7, P8, P11, P15} , where K = 2*4 – 1 = 2* √N – 1 = O(√N)

Maekawa's algorithm –

On initialization

state := RELEASED;
voted := FA□LSE;

For p_i to enter the critical section

state := WANTED;
Multicast *request* to all processes in V_i;
Wait until (number of replies received = K);
state := HELD;

On receipt of a request from p_i at p_j

if (*state* = HELD *or voted* = TRUE)
then
queue *request* from p_i without replying;
else
send *reply* to p_i;
voted := TRUE;
end if

For p_i to exit the critical section

state := RELEASED;
Multicast *release* to all processes in V_i;

On receipt of a release from p_i at p_j

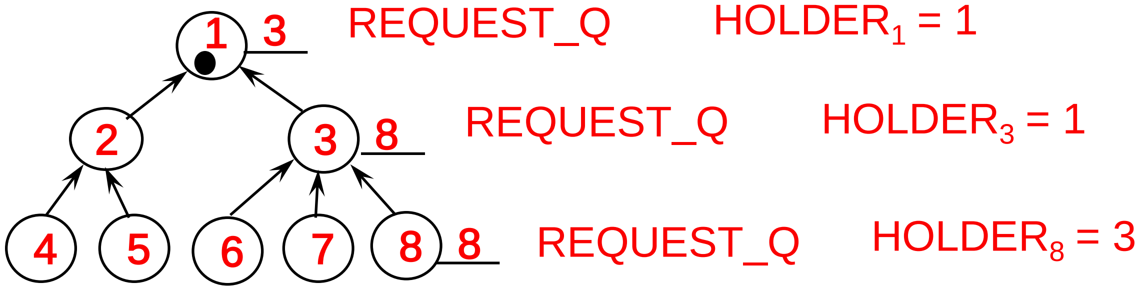
if (queue of requests is non-empty)
then
remove head of queue – from p_k, say;
send *reply* to p_k;
voted := TRUE;
else
voted := FALSE;
end if

Maekawa's Algorithm - Analysis

- **2√N messages per entry, √N messages per exit**
  - **Better than Ricart and Agrawala's (2(N-1) and N-1 messages)**
- **Client delay:**
  - **One round trip time**
- **Synchronization delay:**
  - **One round-trip time**

Raymond's Token-based Approach

- Processes are organized as an un-rooted *n*-ary tree.
- Each process has a variable HOLDER, which indicates the location of the token relative to the node itself.
- Each process keeps a REQUEST_Q that holds the names of neighbors or itself that have sent a REQUEST, but have not yet been sent the token in reply.
-

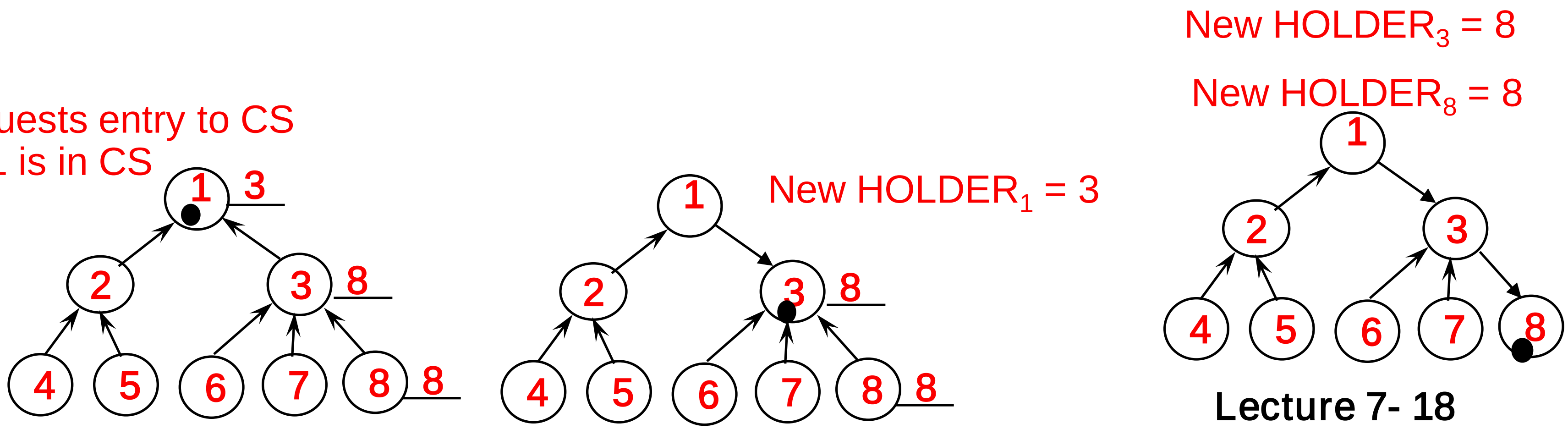


Raymond's Token-based Approach

- **To enter the CS**
  - **Enqueue self.**
  - **If a request has not been sent to HOLDER, send a request.**

- **Upon receipt of a REQUEST message from neighbor x**
  - **If x is not in queue, enqueue x.**
  - **If self is a HOLDER and still in the CS, do nothing further.**
  - **If self is a HOLDER but exits the CS, then get the oldest requester (i.e., dequeue REQUEST_Q), set it to be the new HOLDER, and send token to the new HOLDER.**

P8 requests entry to CS
And P1 is in CS

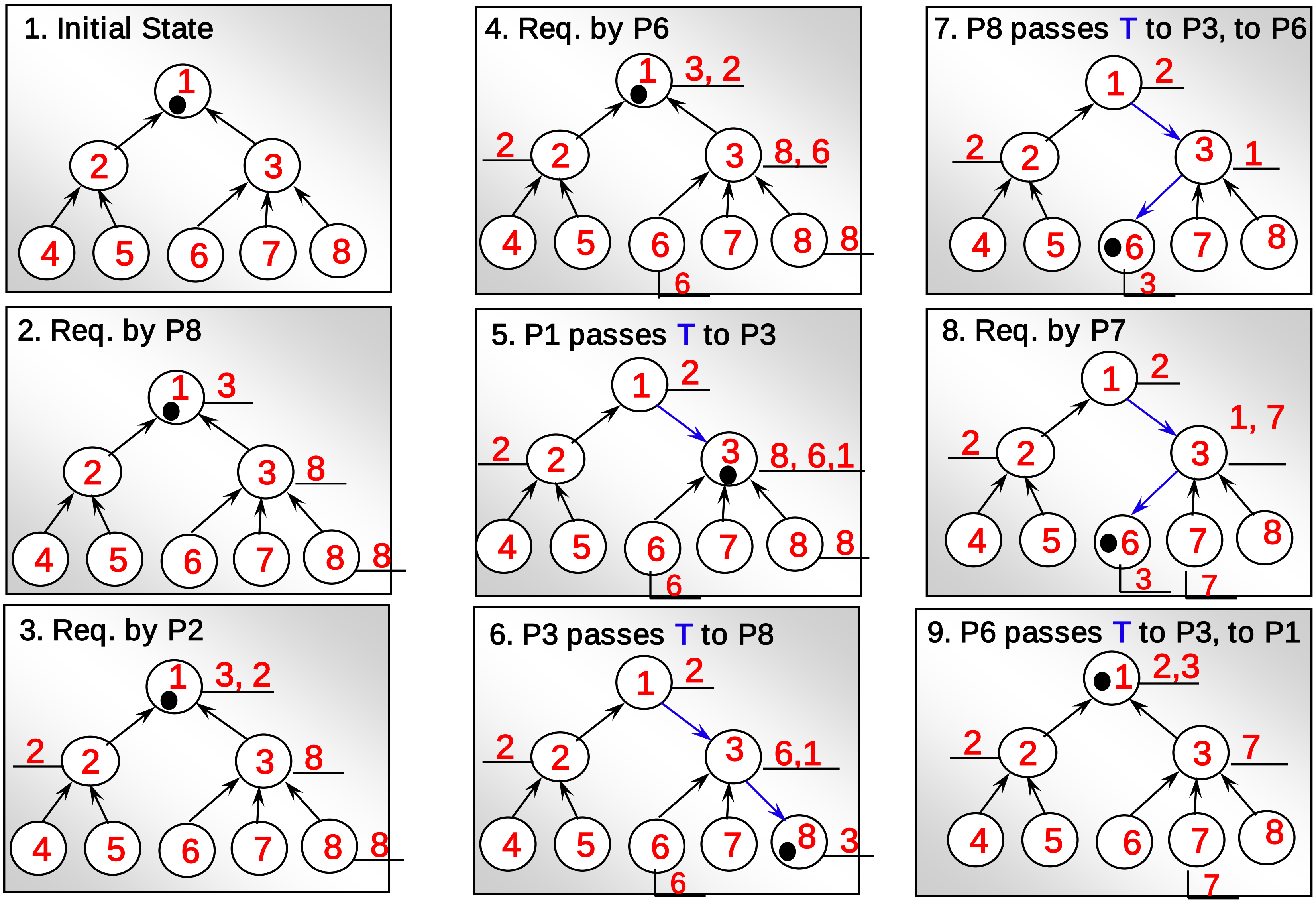


Raymond's Token-based Approach

- **Upon receipt of a token message**
  - **Dequeue REQUEST_Q and set the oldest requester to be HOLDER.**
  - **If HOLDER=self, then hold the token and enter the CS.**
  - **If HOLDER= some other process, send token to HOLDER. In addition, if the (remaining) REQUEST_Q is non-empty, send REQUEST to HOLDER as well.**
- **On exiting the CS**
  - **If REQUEST_Q is empty, continue to hold token.**
  - **If REQUEST_Q is non-empty, then**
    - **dequeue REQUEST_Q and set the oldest requester to HOLDER, and send token to HOLDER.**
    - **In addition, if the (remaining) REQUEST_Q is non-empty, send REQUEST to HOLDER as well.**

Help other Processes access resource transitively

Example:Raymond's Token-based Approach



Analysis

- **Bandwidth?**
- **Client delay?**
- **Synchronization delay?**

- **Try it at home, may be a question for next homework (or an exam question)**

- **Source: K. Raymond, "*A Tree-based algorithm for distributed mutual exclusion*", ACM Transactions on Computer Systems (TOCS), 7(1): 61-77, 1989**

Summary

- **Mutual exclusion**
  - **Coordinator-based token**
  - **Token ring**
  - **Ricart and Agrawala’s timestamp algorithm**
  - **Maekawa’s algorithm**
  - **Raymond’s algorithm**