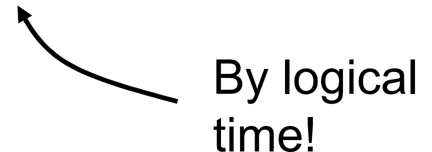
Distributed Mutual Exclusion

Last time…

- Synchronizing real, distributed clocks
- Logical time and concurrency
- Lamport clocks and total-order Lamport clocks

Goals of distributed mutual exclusion

- Much like regular mutual exclusion
  - Safety: mutual exclusion
  - Liveness: progress
  - Fairness: bounded wait and in-order
- Secondary goals:
  - reduce message traffic
  - minimize synchronization delay
    - i.e., switch quickly between waiting processes



Distributed mutex is different

- Regular mutual exclusion solved using shared state, e.g.
  - atomic test-and-set of a shared variable…
  - shared queue…
- We solve distributed mutual exclusion with message passing
  - Note: we assume the network is reliable but asynchronous…but processes might fail!

Solution 1: A central mutex server

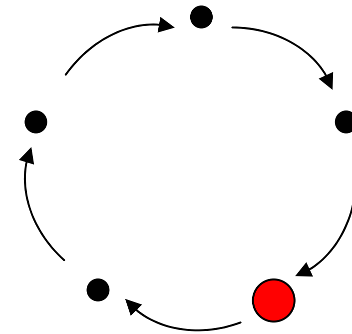
- To enter critical section:
  - send REQUEST to central server, wait for permission
- To leave:
  - send RELEASE to central server

Solution 1: A central mutex server

- Advantages:
  - Simple (we like simple!)
  - Only 3 messages required per entry/exit
- Disadvantages:
  - Central point of failure
  - Central performance bottleneck
  - With an asynchronous network, impossible to achieve in-order fairness
  - Must elect/select central server

Solution 2: A ring-based algorithm

- Pass a token around a ring
  - Can enter critical section only if you hold the token
- Problems:
  - Not in-order
  - Long synchronization delay
    - Need to wait for up to *N-1* messages, for *N* processors
  - Very unreliable
    - Any process failure breaks the ring



2’: A fair ring-based algorithm

- Token contains the time *t* of the earliest known outstanding request
- To enter critical section:
  - Stamp your request with the current time T_r, wait for token
- When you get token with time *t* while waiting with request from time T_r, compare T_r to *t*:
  - If $T_r = t$: hold token, run critical section
  - If $T_r > t$: pass token
  - If *t* not set or $T_r < t$: set token-time to T_r, pass token, wait for token
- To leave critical section:
  - Set token-time to null (i.e., unset it), pass token

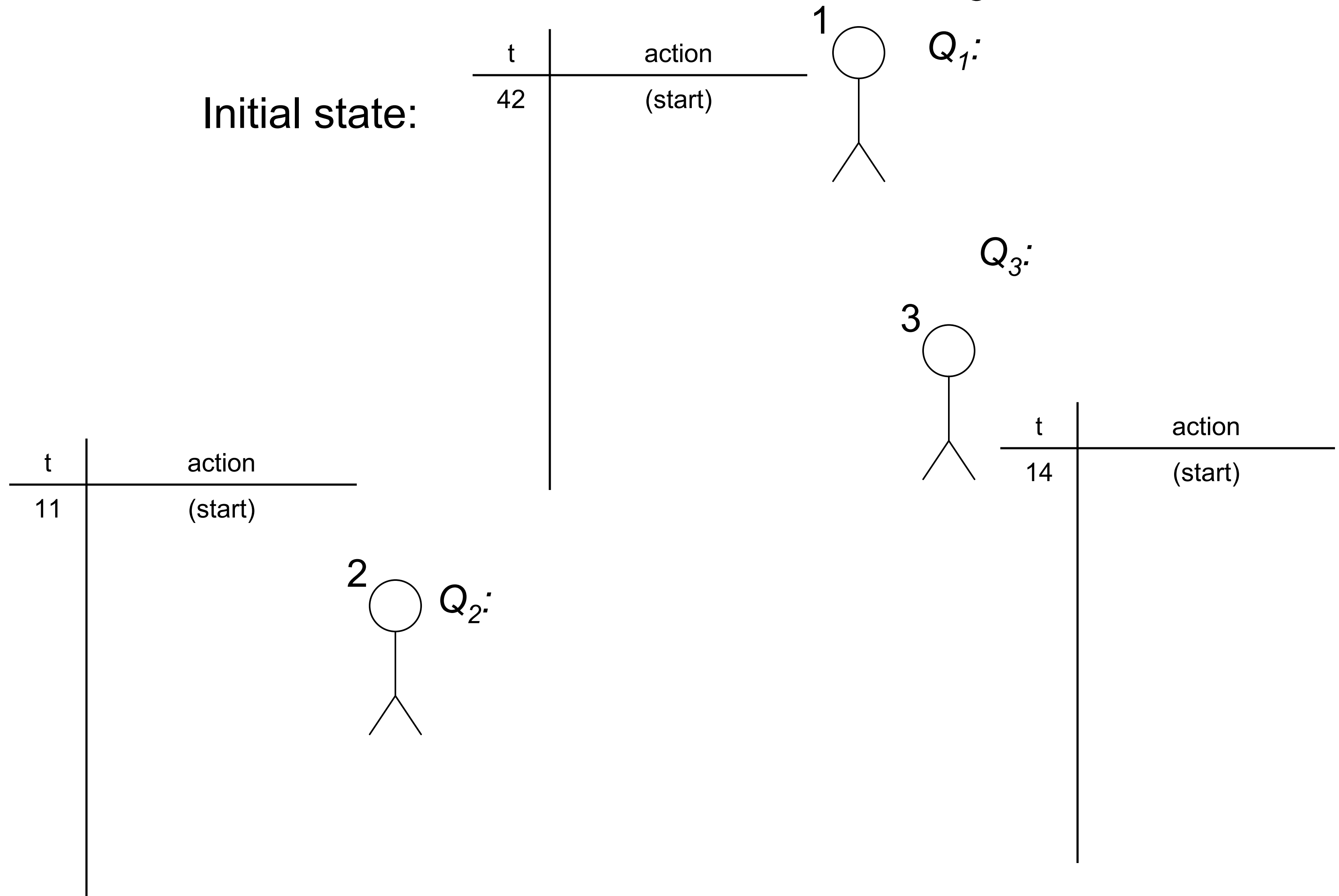
Solution 3: A shared priority queue

- By Lamport, using Lamport clocks
- Each process *i* locally maintains Q_i, part of a shared priority queue
- To run critical section, must have replies from all other processes AND be at the front of Q_i
  - When you have all replies:
    - #1: All other processes are aware of your request
    - #2: You are aware of any earlier requests for the mutex

Solution 3: A shared priority queue

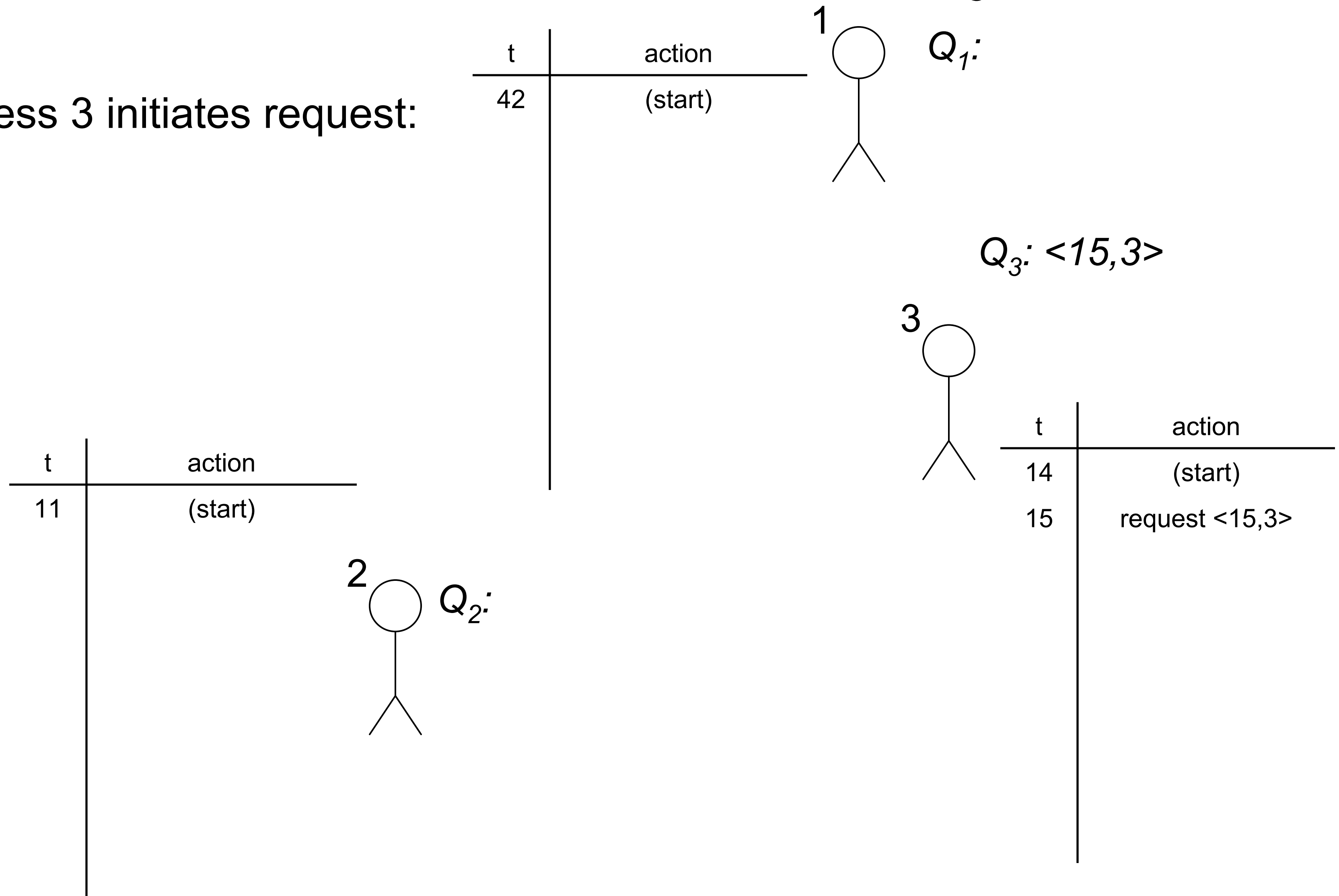
- To enter critical section at process *i* :
  - Stamp your request with the current time *T*
  - Add request to Q_i
  - Broadcast REQUEST(*T*) to all processes
  - Wait for all replies and for *T* to reach front of Q_i
- To leave:
  - Pop head of Q_i, Broadcast RELEASE to all processes
- On receipt of REQUEST(*T'*) from process *j*:
  - Add *T'* to Q_i
  - If waiting for REPLY from *j* for an earlier request *T*, wait until *j* replies to you
  - Otherwise REPLY
- On receipt of RELEASE
  - Pop head of Q_i

Solution 3: A shared priority queue



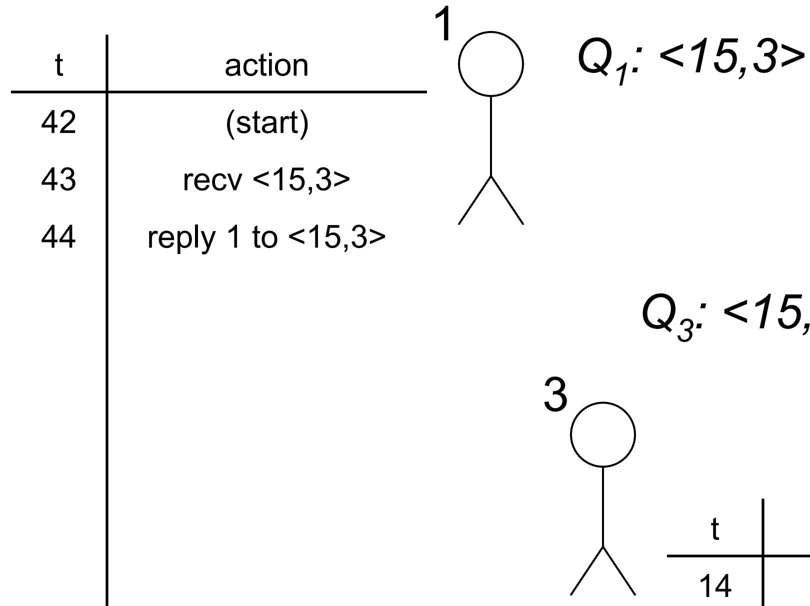
Solution 3: A shared priority queue

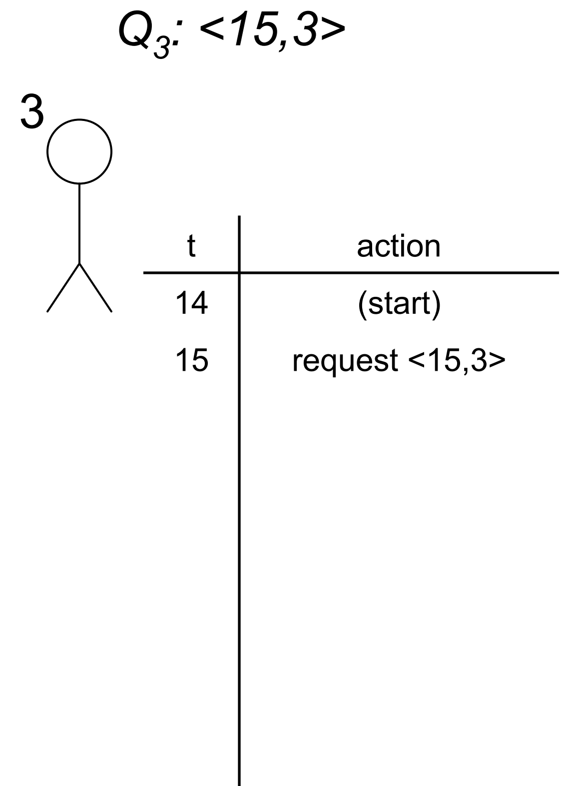
Process 3 initiates request:

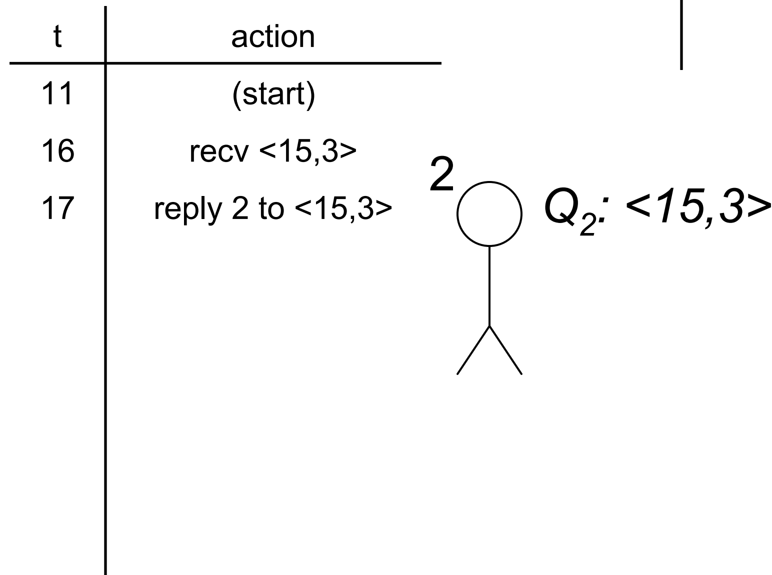


Solution 3: A shared priority queue

1 & 2 receive and reply

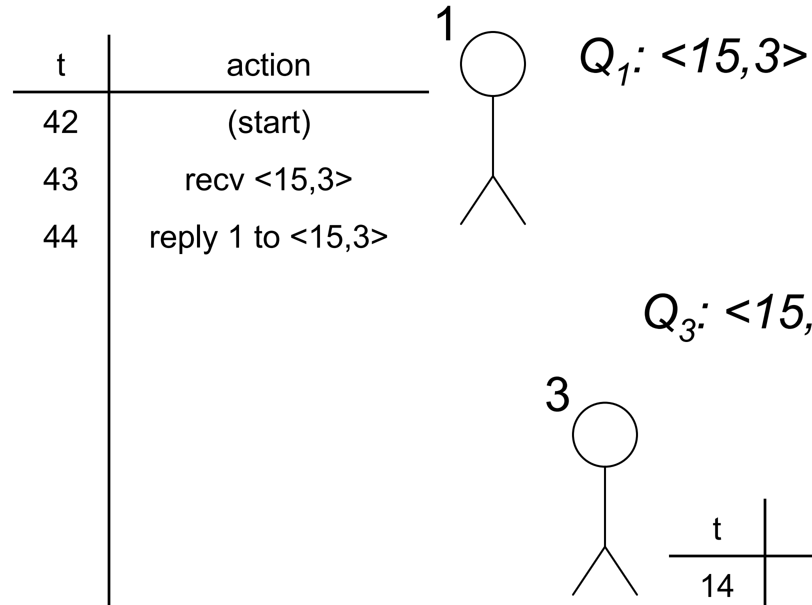


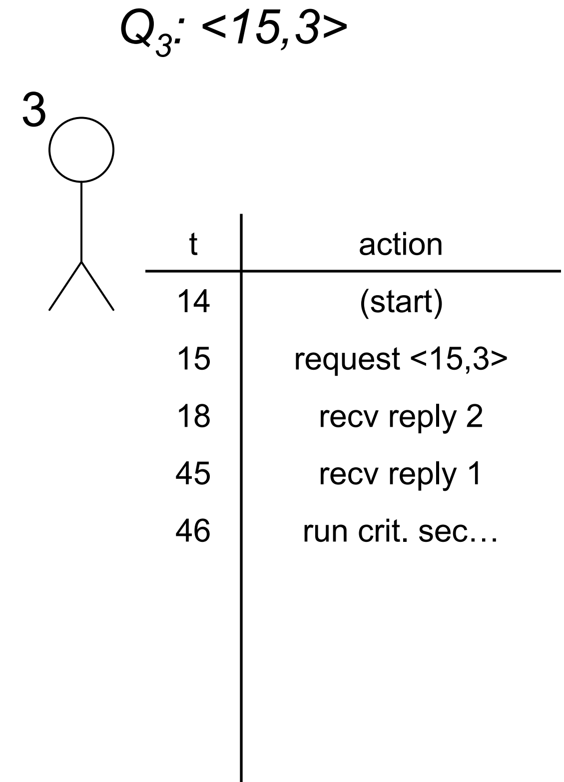


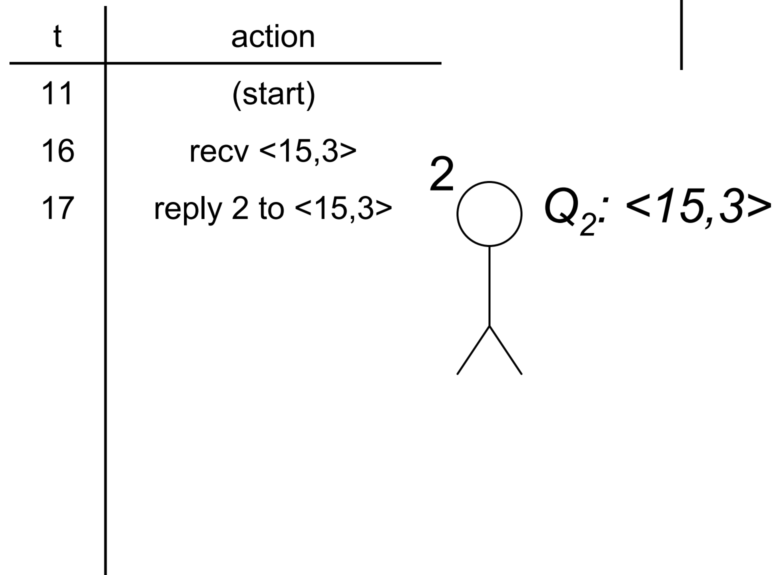


Solution 3: A shared priority queue

3 gets replies, is on front of queue, can run crit. section:

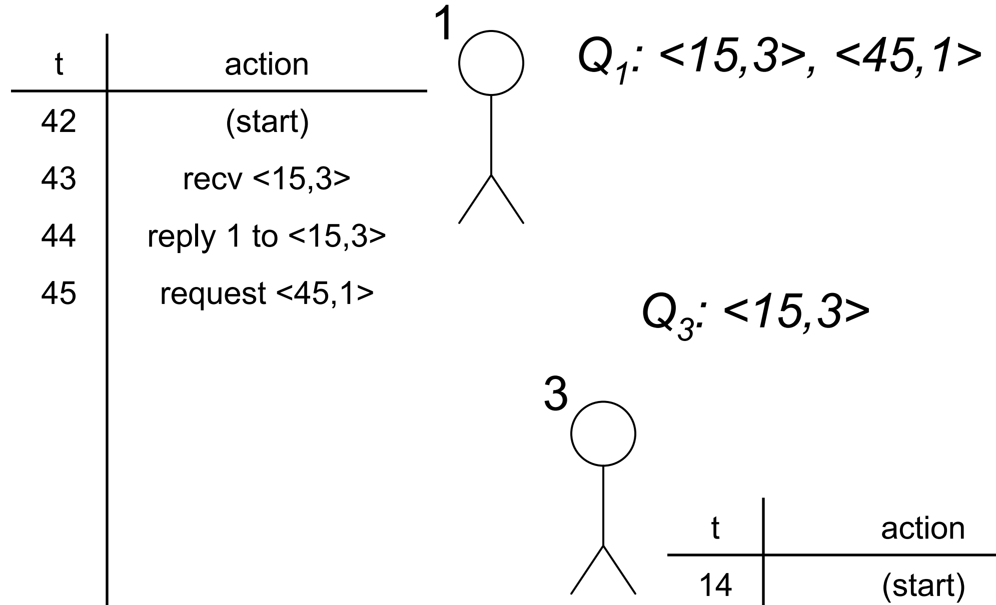


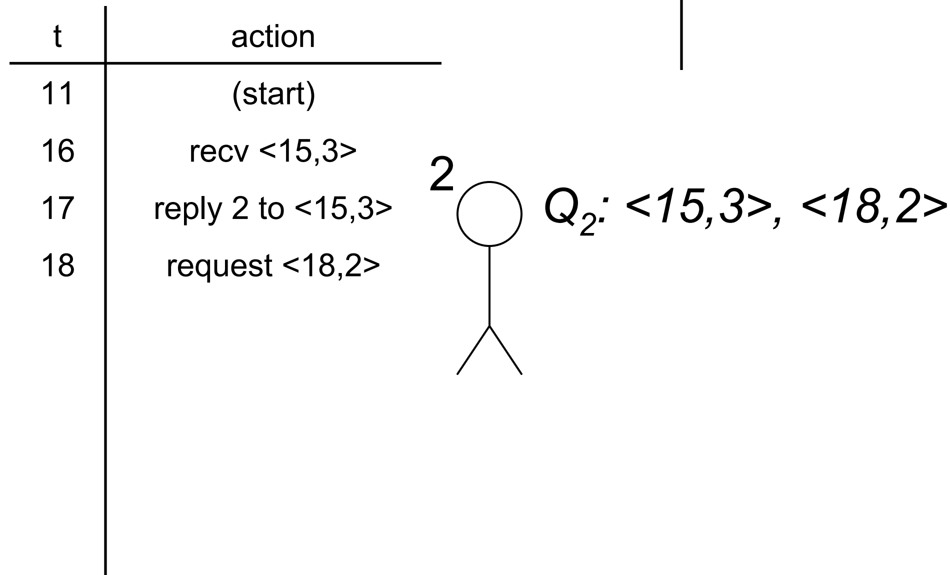


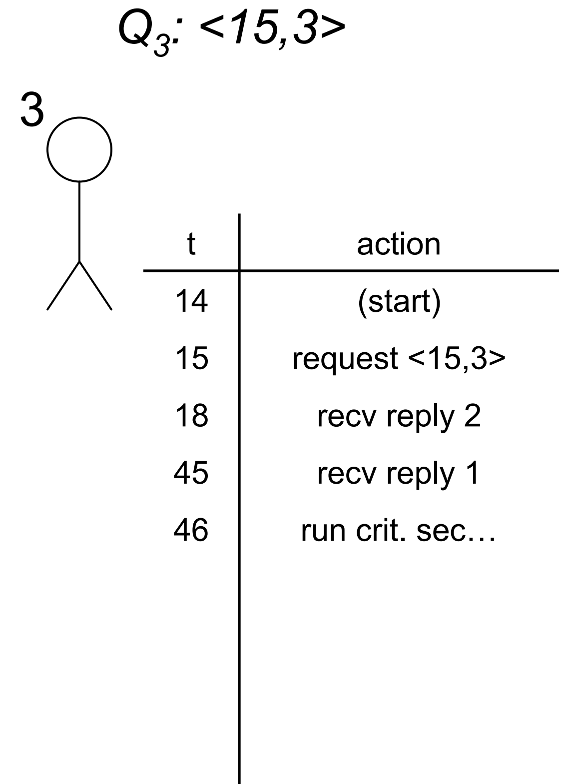


Solution 3: A shared priority queue

Processes 1 and 2 concurrently initiate requests:

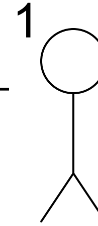






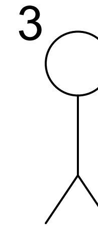
Solution 3: A shared priority queue

Process 3 gets requests and replies:



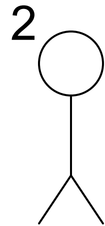
Q_1: <15,3>, <45,1>

t	action
42	(start)
43	recv <15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3



Q_3: <15,3>, <18,2>, <45,1>

t	action
14	(start)
15	request <15,3>
18	recv reply 2
45	recv reply 1
46	run crit. sec…
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>



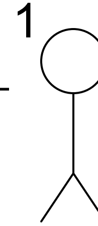
Q_2: <15,3>, <18,2>

t	action
11	(start)
16	recv <15,3>
17	reply 2 to <15,3>
18	request <18,2>
51	recv reply 3

Solution 3: A shared priority queue

Process 2 gets request <45,1>, delays reply because <18,2> is an earlier request to which Process 1 has not replied

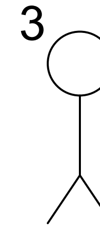
Process 1:



Q_1: *<15,3>, <45,1>*

t	action
42	(start)
43	recv <15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3

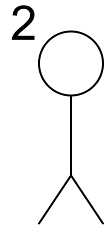
Process 3:



Q_3: *<15,3>, <18,2>, <45,1>*

t	action
14	(start)
15	request <15,3>
18	recv reply 2
45	recv reply 1
46	run crit. sec…
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

Process 2:

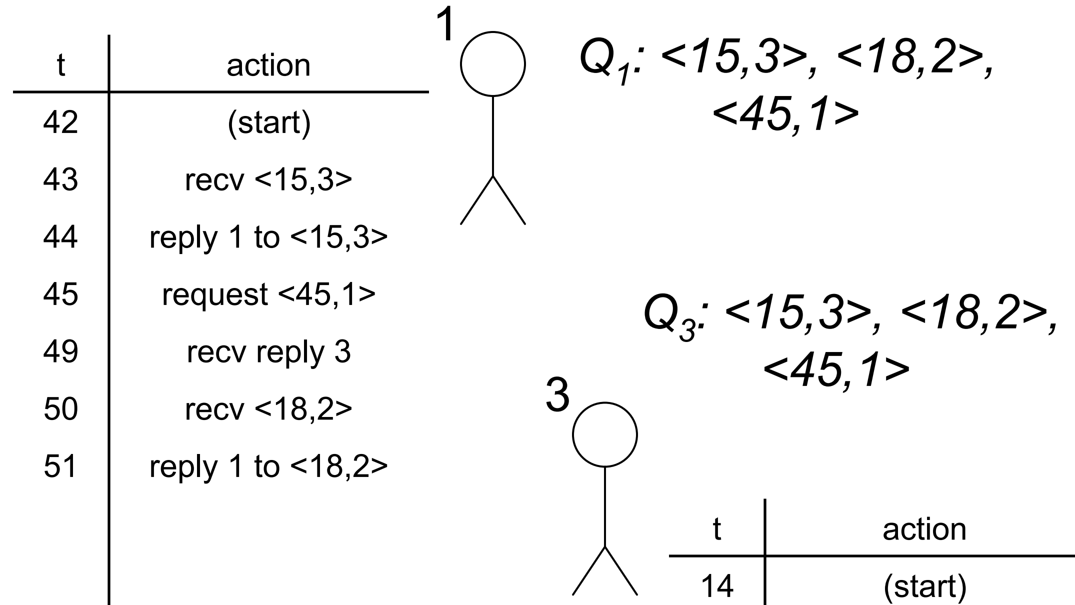


Q_2: *<15,3>, <18,2>, <45,1>*

t	action
11	(start)
16	recv <15,3>
17	reply 2 to <15,3>
18	request <18,2>
51	recv reply 3
52	recv <45,1>

Solution 3: A shared priority queue

Process 1 gets request <18,2>, replies



1

Q_1: <15,3>, <18,2>, <45,1>

t	action
42	(start)
43	recv <15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3
50	recv <18,2>
51	reply 1 to <18,2>

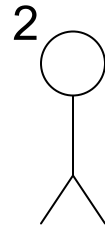
3

Q_3: <15,3>, <18,2>, <45,1>

t	action
14	(start)
15	request <15,3>
18	recv reply 2
45	recv reply 1
46	run crit. sec…
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

t	action
11	(start)
16	recv <15,3>
17	reply 2 to <15,3>
18	request <18,2>
51	recv reply 3
52	recv <45,1>

2

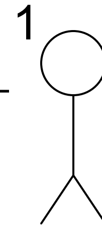


Q_2: <15,3>, <18,2>, <45,1>

Solution 3: A shared priority queue

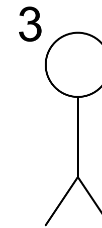
Process 2 gets reply from process 1, finally replies to <45,1>

t	action
42	(start)
43	recv <15,3>
44	reply 1 to <15,3>
45	request <45,1>
49	recv reply 3
50	recv <18,2>
51	reply 1 to <18,2>



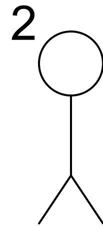
Q_1: <15,3>, <18,2>, <45,1>

Q_3: <15,3>, <18,2>, <45,1>



t	action
14	(start)
15	request <15,3>
18	recv reply 2
45	recv reply 1
46	run crit. sec…
47	recv <45,1>
48	reply 3 to <45,1>
49	recv <18,2>
50	reply 3 to <18,2>

t	action
11	(start)
16	recv <15,3>
17	reply 2 to <15,3>
18	request <18,2>
51	recv reply 3
52	recv <45,1>
53	recv reply 1
54	reply 2 to <45,1>



Q_2: <15,3>, <18,2>, <45,1>

Solution 3: A shared priority queue

- Advantages:
  - Fair
  - Short synchronization delay
- Disadvantages:
  - Very unreliable
    - Any process failure halts progress
  - *3(N-1)* messages per entry/exit

Solution 4: Ricart and Agrawala

- An improved version of Lamport's shared priority queue
  - Combines function of REPLY and RELEASE messages
- Delay REPLY to any requests later than your own
  - Send all delayed replies after you exit your critical section

Solution 4: Ricart and Agrawala

- To enter critical section at process *i* :
  - Same as Lamport's algorithm
    - Except you don't need to reach the front of Q_i to run your critical section: you just need all replies
- To leave:
  - Broadcast REPLY to all processes in Q_i
  - Empty Q_i
- On receipt of REQUEST(*T'*):
  - If waiting for (or in) critical section for an earlier request *T*, add *T'* to Q_i
  - Otherwise REPLY immediately

Ricart and Agrawala safety

- Suppose request T_1 is earlier than T_2. Consider how the process for T_2 collects its reply from process for T_1:
  - T_1 must have already been time-stamped when request T_2 was received, otherwise the Lamport clock would have been advanced past time T_2
  - But then the process must have delayed reply to T_2 until after request T_1 exited the critical section. Therefore T_2 will not conflict with T_1.

Solution 4: Ricart and Agrawala

- Advantages:
  - Fair
  - Short synchronization delay
  - Better than Lamport's algorithm
- Disadvantages
  - Very unreliable
  - *2(N-1)* messages for each entry/exit

Solution 5: Majority rules

- Instead of collecting REPLYs, collect VOTEs
  - Each process VOTEs for which process can hold the mutex
  - Each process can only VOTE once at any given time
  - You hold the mutex if you have a majority of the VOTEs
    - Only possible for one process to have a majority at any given time!

Solution 5: Majority rules

- To enter critical section at process *i* :
  - Broadcast REQUEST(T), collect VOTEs
  - Can enter crit. sec. if collect a majority of VOTEs
- To leave:
  - Broadcast RELEASE-VOTE to all processes who VOTEd for you
- On receipt of REQUEST(*T'*) from process *j*:
  - If you have not VOTEd, VOTE for *T'*
    - Otherwise, add *T'* to Q_i
- On receipt of RELEASE-VOTE*:*
  - If Q_i not empty, VOTE for pop(Q_i)

Solution 5: Majority rules

- Advantages:
  - Can progress with as many as *N/2 – 1* failed processes
- Disadvantages:
  - Not fair
  - Deadlock!
    - No guarantee that anyone receives a majority of votes

Solution 5': Dealing with deadlock

- Allow processes to ask for their vote back
  - If already VOTEd for *T'* and get a request for an earlier request *T*, RESCIND-VOTE for *T'*
  - If receive RESCIND-VOTE request and not in critical section, RELEASE-VOTE and re-REQUEST
- Guarantees that some process will eventually get a majority of VOTEs
- Still not fair…

Solution 6: Maekawa voting

- Each process *i* has an associated voting set of other processes, V_i
- To get mutex, need VOTE from all of V_i
  - As long as $V_i \cap V_j$ is non-empty for all *i,j*, no two processes will hold mutex at the same time
- E.g, arrange processes in a 2-dimensional grid
  - Let V_i be row and column of *i*

Solution 6: Maekawa voting

- 2-dimensional grid requires ~ 2 sqrt(N) votes, for ~ 6 sqrt(N) messages
  - More complex Maekawa solutions require only ~ sqrt(N) votes
- Can deadlock, not fair
  - RESCIND-VOTE solution can fix deadlock…
- Unreliable
  - Any failure in your voting sets prevents you from getting the mutex