Mutual exclusion

Example use I – mutual exclusion [access granted in order]

Goal: a solution for granting a resource to a process which satisfies the following three conditions:

- [mutual exclusion] A process which has been granted the resource must release it before the resource can be granted to another process.
- [termination/progress] If every process which is granted the resource eventually releases it, then every request is eventually granted.
- [ordering] Requests must be granted in the order in which they are made.

Distributed Mutual Exclusion Is Different

Regular mutual exclusion solved using shared state

– E.g., atomic test-and-set of shared variable

Only tool for distributed mutual exclusion:

message passing

Assumptions

- Communication channels
  - reliable: messages do arrive at destination
  - in-order: messages are delivered in the order in which they are sent
  - asynchronous (messages may take long time)
- Participants do not fail.

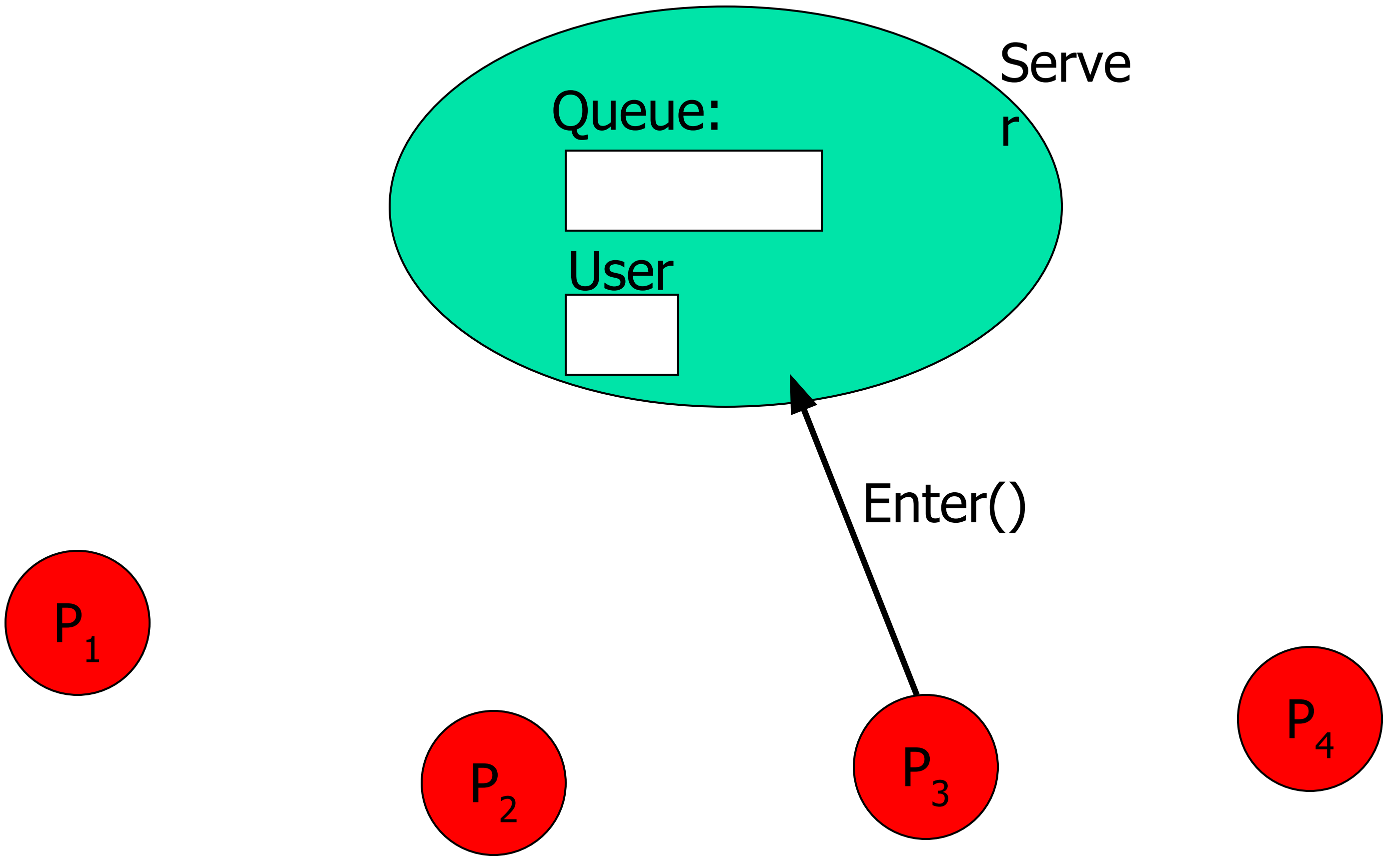
Q: Potential Designs?

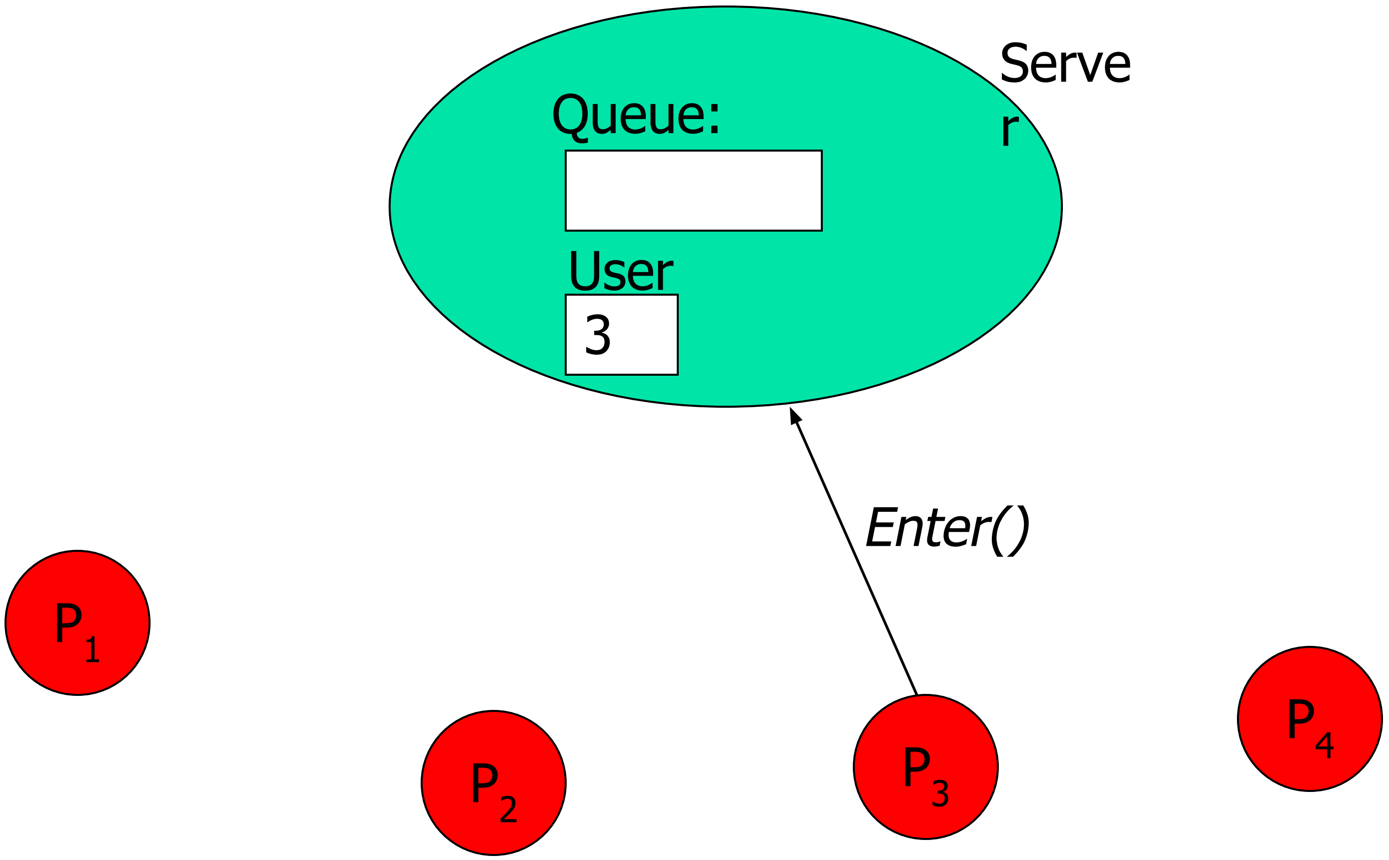
Distributed Mutual Exclusion Protocols

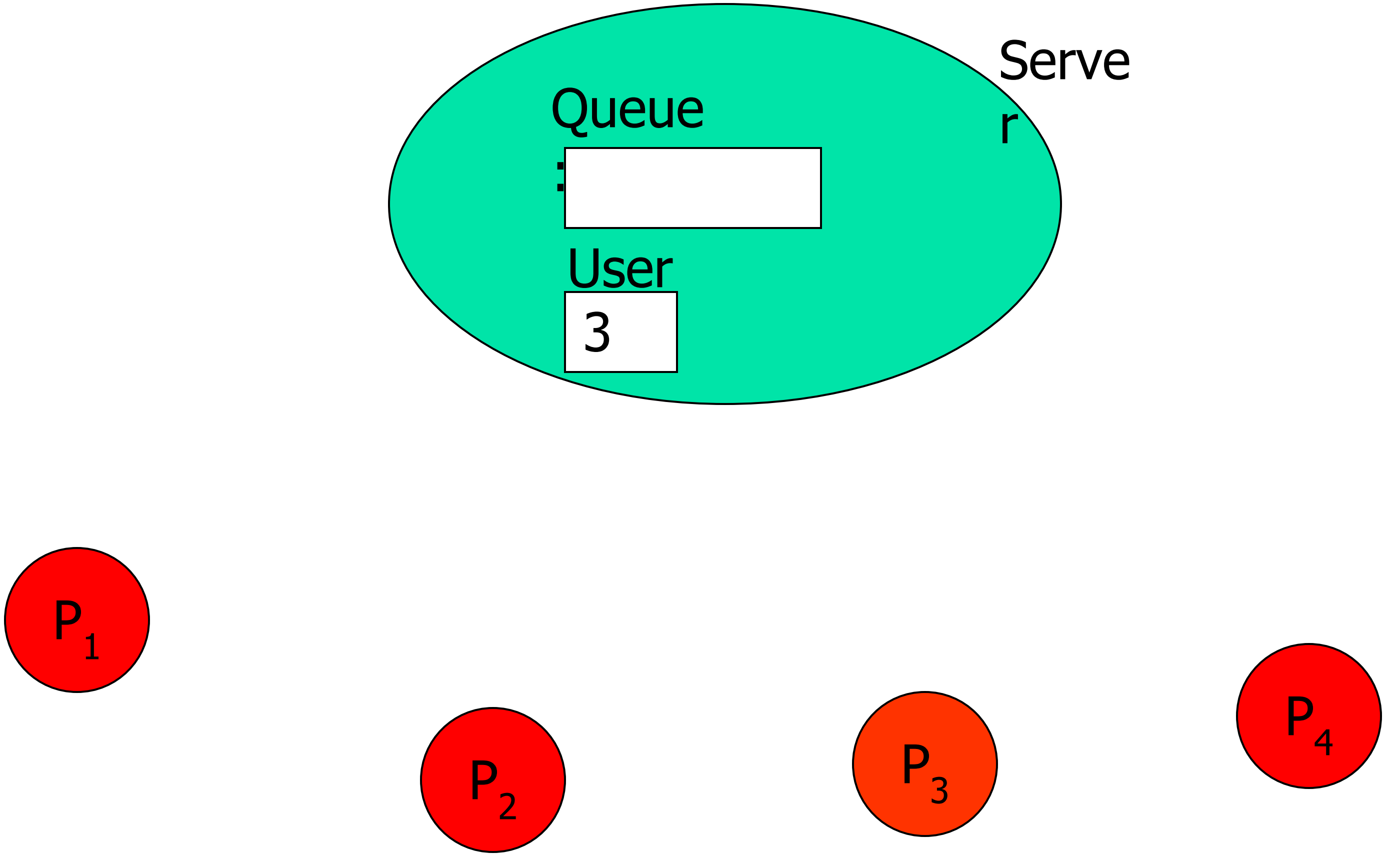
- Key ideas:
  - Before entering critical section, processor must get permission (from other processors, or from arbitrator)
  - When exiting critical section, processor must let the others know that he’s finished
  - For fairness, processors allow other processors who have asked for permission earlier than them to proceed

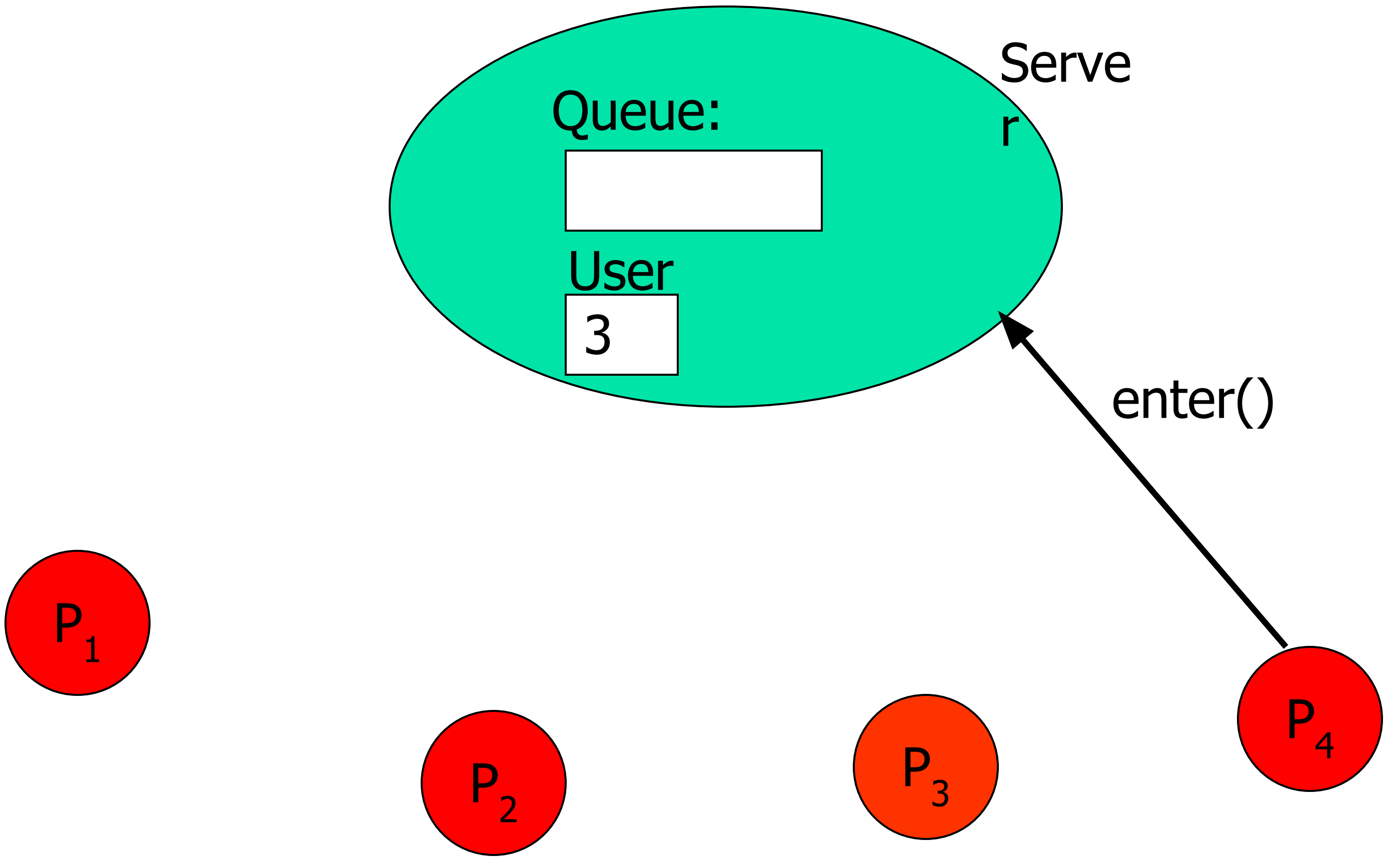
Solution 1: Centralized Lock Server

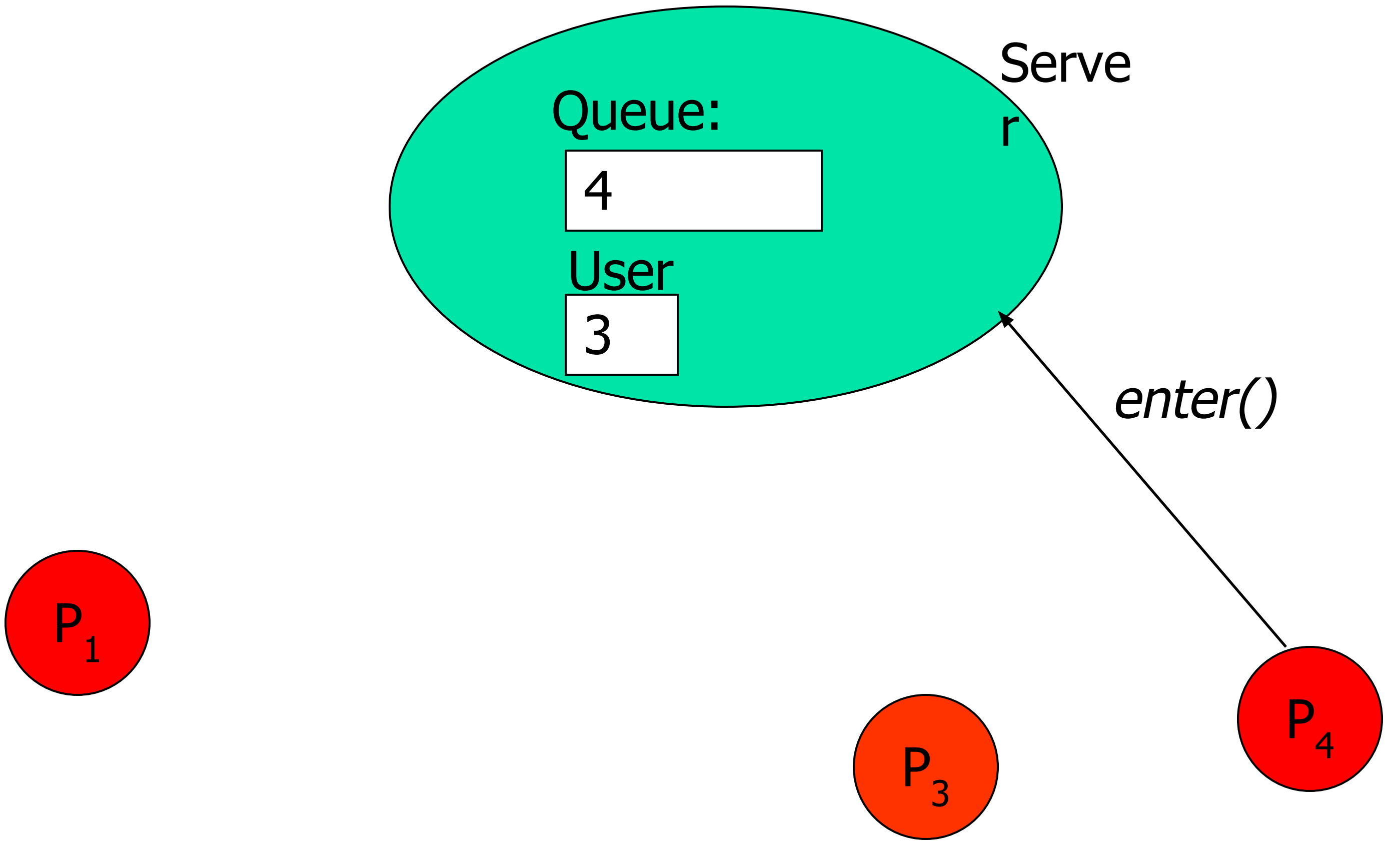
- To enter critical section:
  - send REQUEST to central server
  - wait for permission from server
- To leave critical section:
  - send RELEASE to central server
- Server:
  - Has an internal queue of all REQUESTs it's received
    but to which it hasn't yet sent OK
  - Delays sending OK back to process until process is at head of queue
  - Removes process from the queue after it get RELEASE

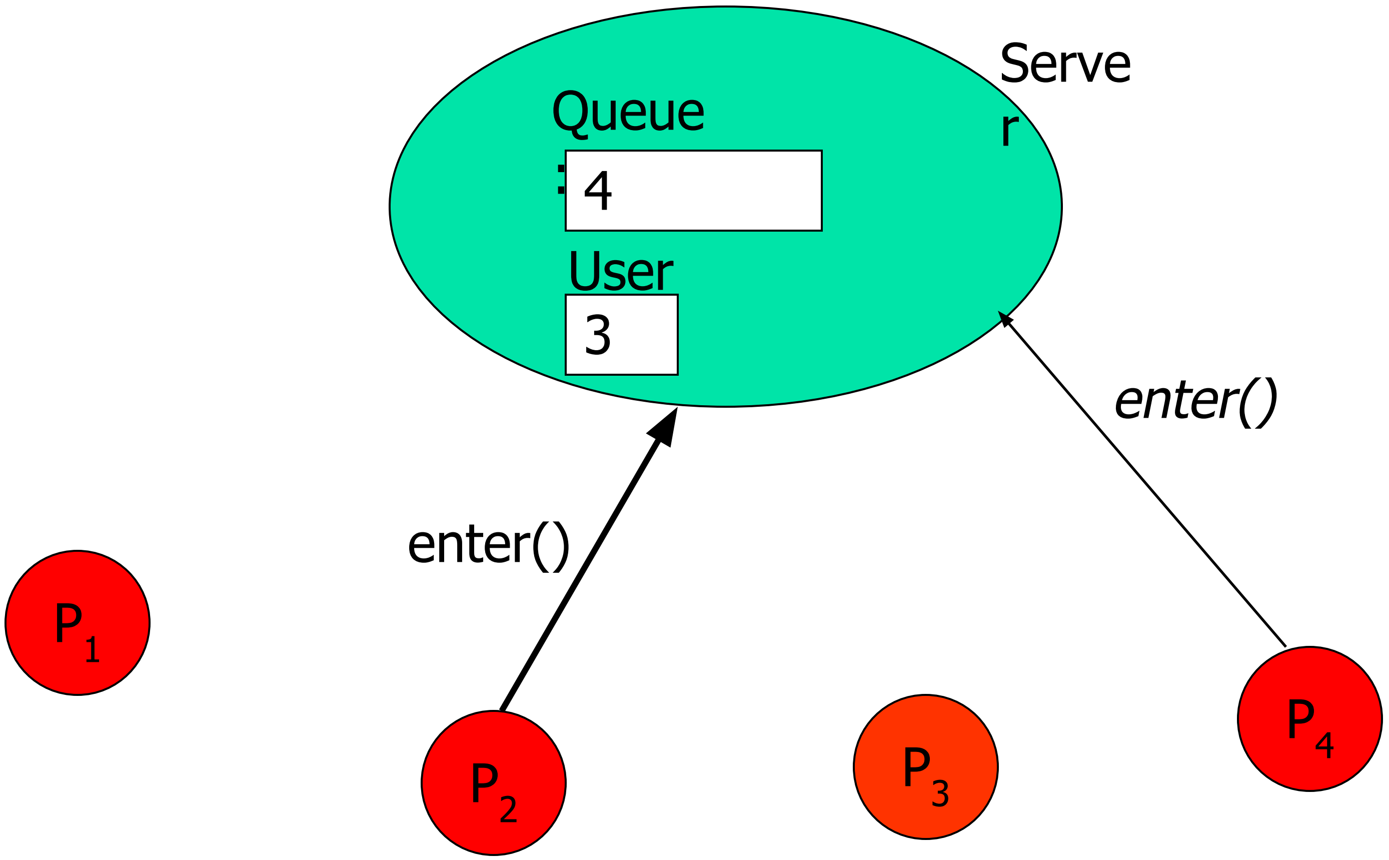

Serve
r
Queue:
User
Enter()
P_1
P_2
P_3
P_4

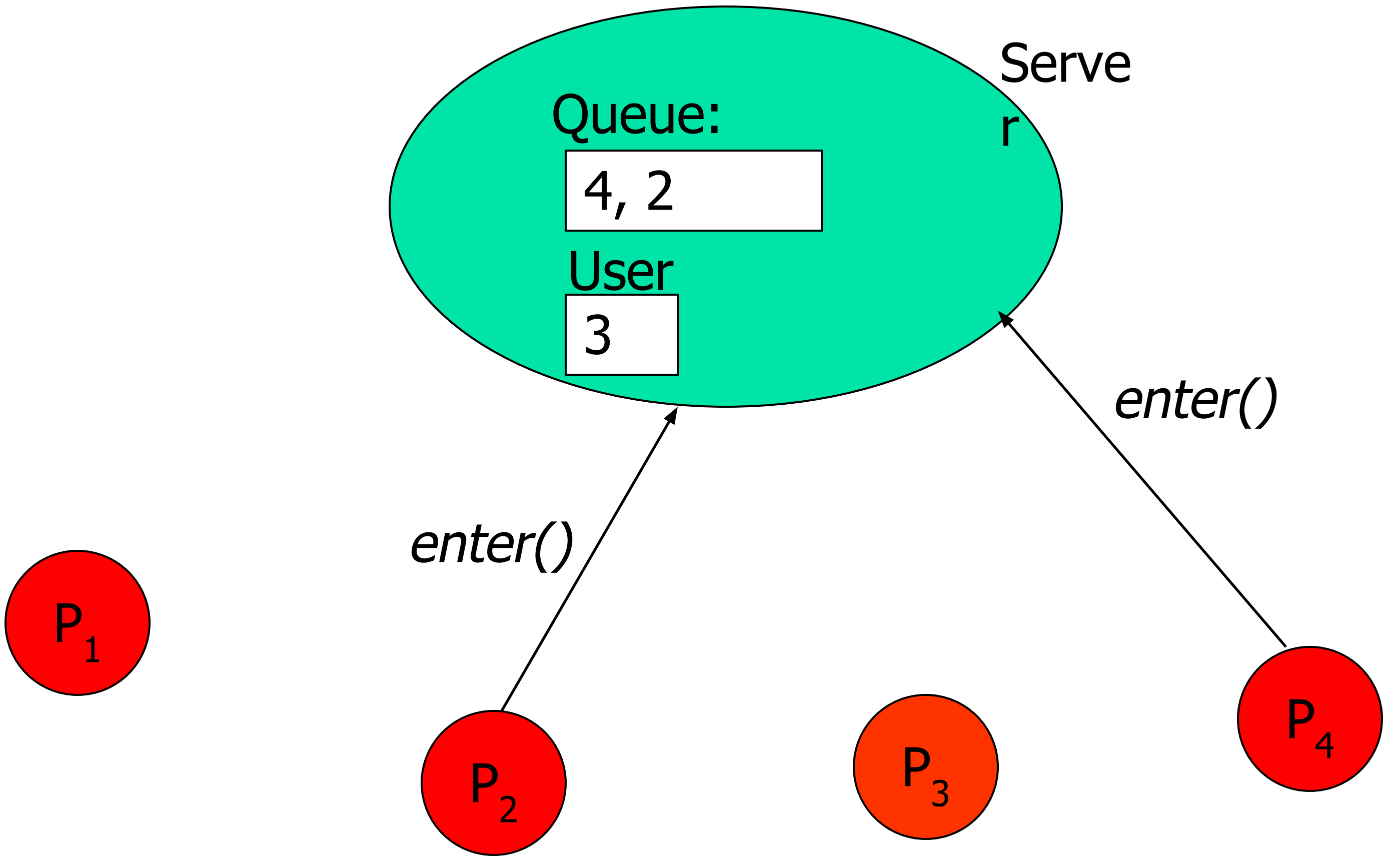

Serve
r
Queue:
User
3
Enter()
P_1
P_2
P_3
P_4

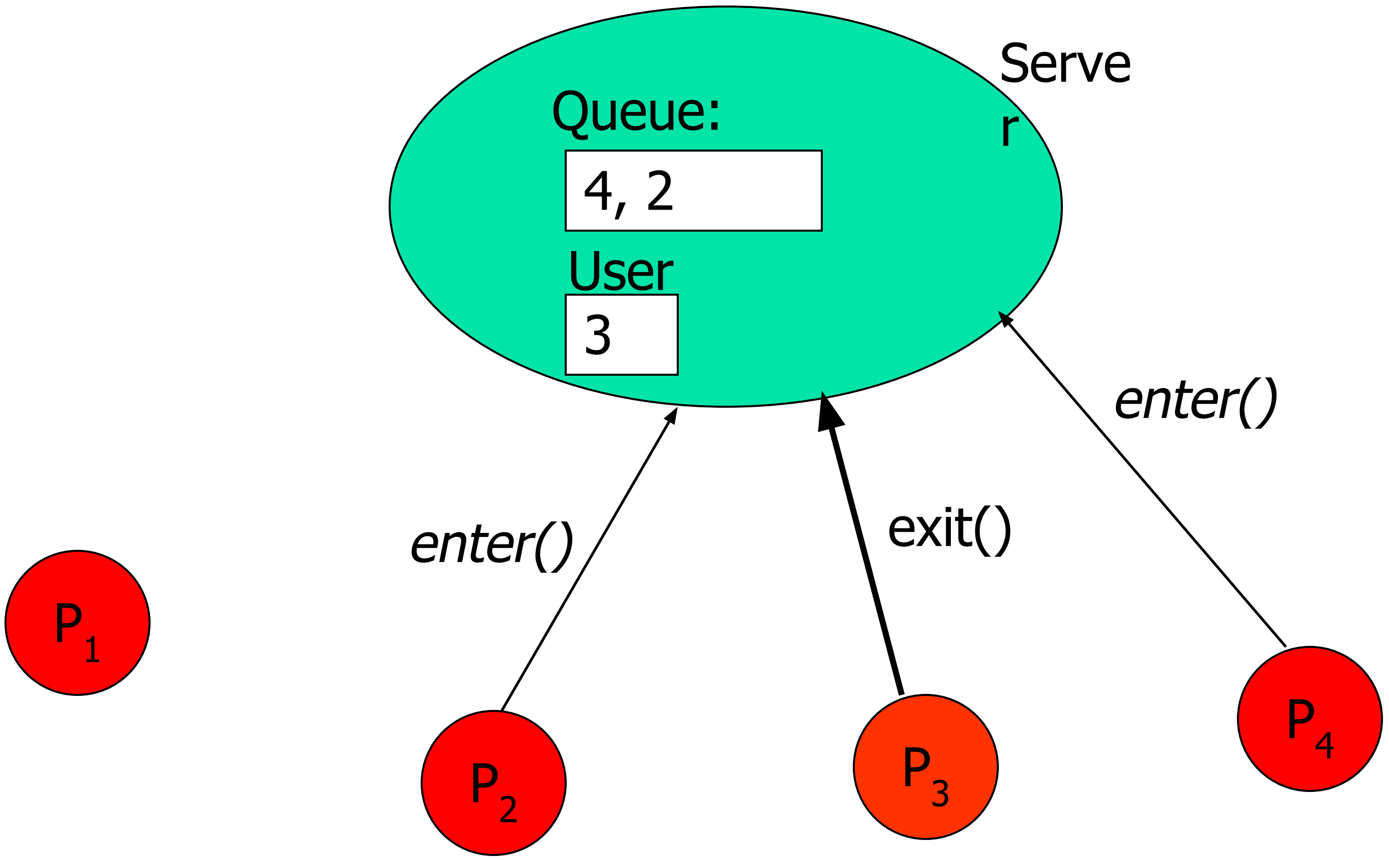

Serve
r
Queue
:
User
3
P_1
P_2
P_3
P_4


Serve
r
Queue:
User
3
enter()
P
1
P
2
P
3
P
4


Serve
r
Queue:
4
User
3
enter()
P_1
P_3
P_4


Server
Queue:
4
User
3
enter()
enter()
P1
P2
P3
P4


Serve
r
Queue:
4, 2
User
3
enter()
enter()
P_1
P_2
P_3
P_4



Serve
r

Queue:

4, 2

User

3

enter()

exit()

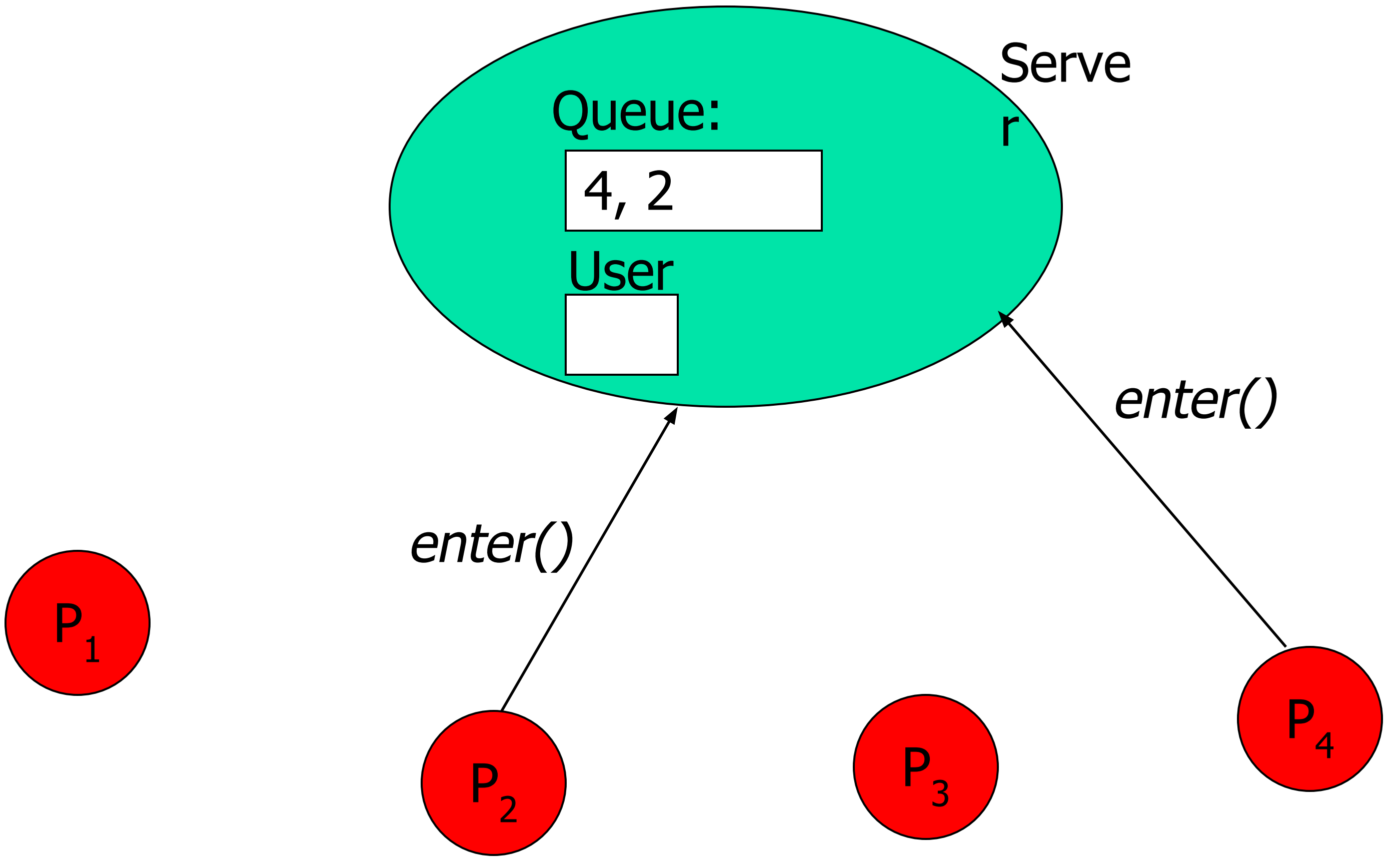
enter()

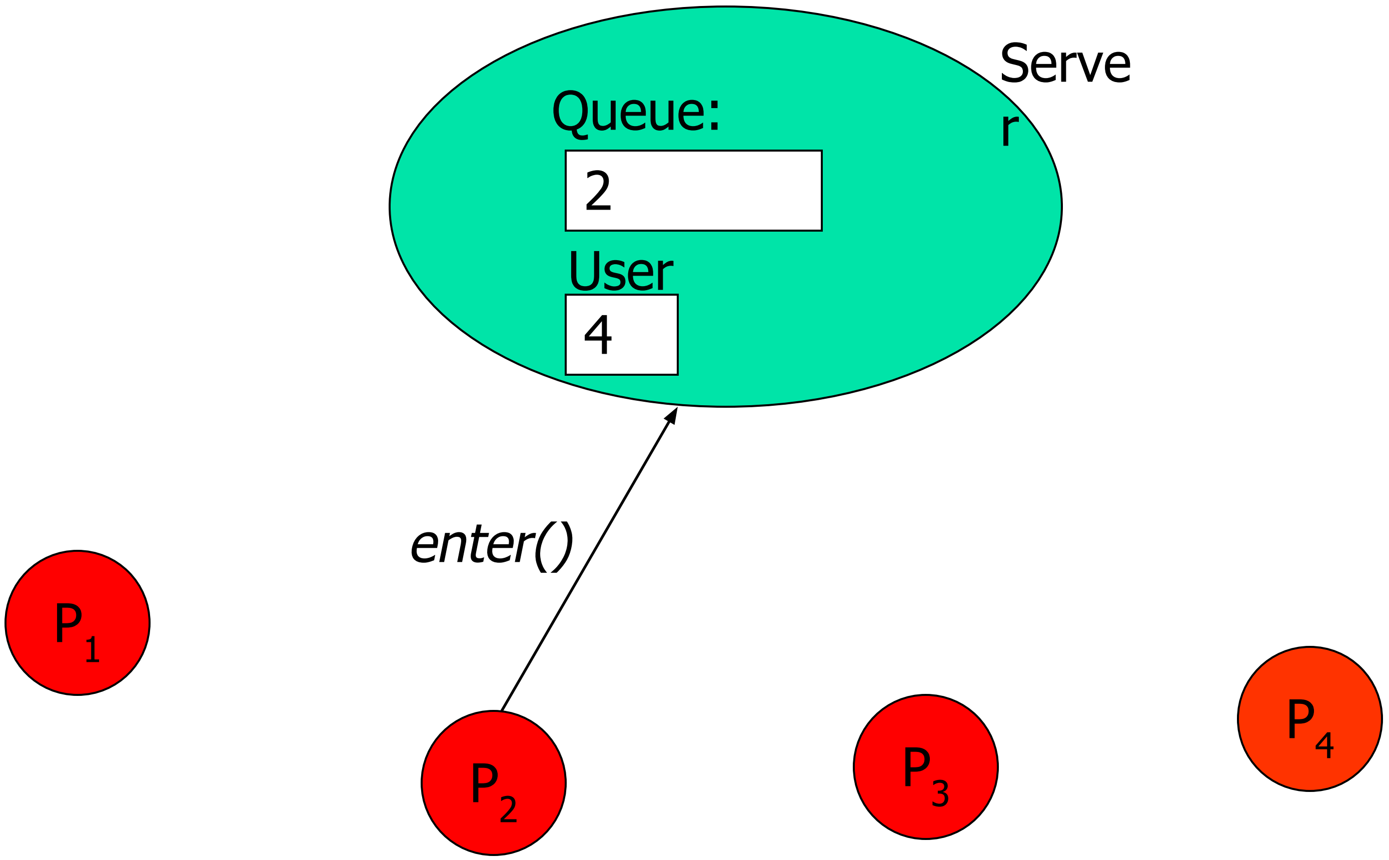
P_1

P_2

P_3

P_4


Serve
r
Queue:
4, 2
User
enter()
enter()
P1
P2
P3
P4



Server

Queue:

2

User

4

enter()

P_1

P_2

P_3

P_4

Lamport's algorithm -

- Each process (locally) maintains *Qi* a local copy of a shared priority queue

- To enter critical section, *P*
  - must have replies from all other processes AND
  - be at the front of *Qi*

- Rationale: When P has all replies:
  - #1: All other processes are aware of P's request
  - #2: P is aware of any earlier requests for the mutex (if any)

Advantages:

– Simple (we like simple!)

– Only 3 messages required per sync session (enter&exit)

Disadvantages:

– Single point of failure

– Single performance bottleneck

– With an asynchronous network, doesn’t achieve in-order fairness (even for logical time order)

– Must select (or *elect*) a central server

Solution 2: A ring-based algorithm

Pass a token around a ring

– Participant enters critical section only if it holds 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

Does not guarantee ordering (but this can be fixed). How?

1. To request the resource: process P^i sends
    1. the message *request_resource* (T^m:P^i) to every process, and
    2. puts that message on its own request queue

   [T^m is the timestamp of resource request: i.e. logical clock when req. is sent]

2. When P^j receives *request_resource* (T^m:P^i),
    - it places it on its request queue
    - sends a (timestamped) acknowledgment message to P^i

4. To release the resource, P^i
    1. removes any *request_resource*(T^m:P^i) message from its request queue
    2. sends a (timestamped) *release_resource* (P^i) to every other process.

5. When process P^j receives a P^i releases resource message,
    1. it removes any *request_resource* (T^m:P^i) message from its queue.

6. Process P^i is granted the resource when
    1. there is a *requests resource* (T^m:P^i) message at the top of its own request queue (ordered by logical timestamp in T^m) AND
    2. P^i has received a message from every other process timestamped later than T^m

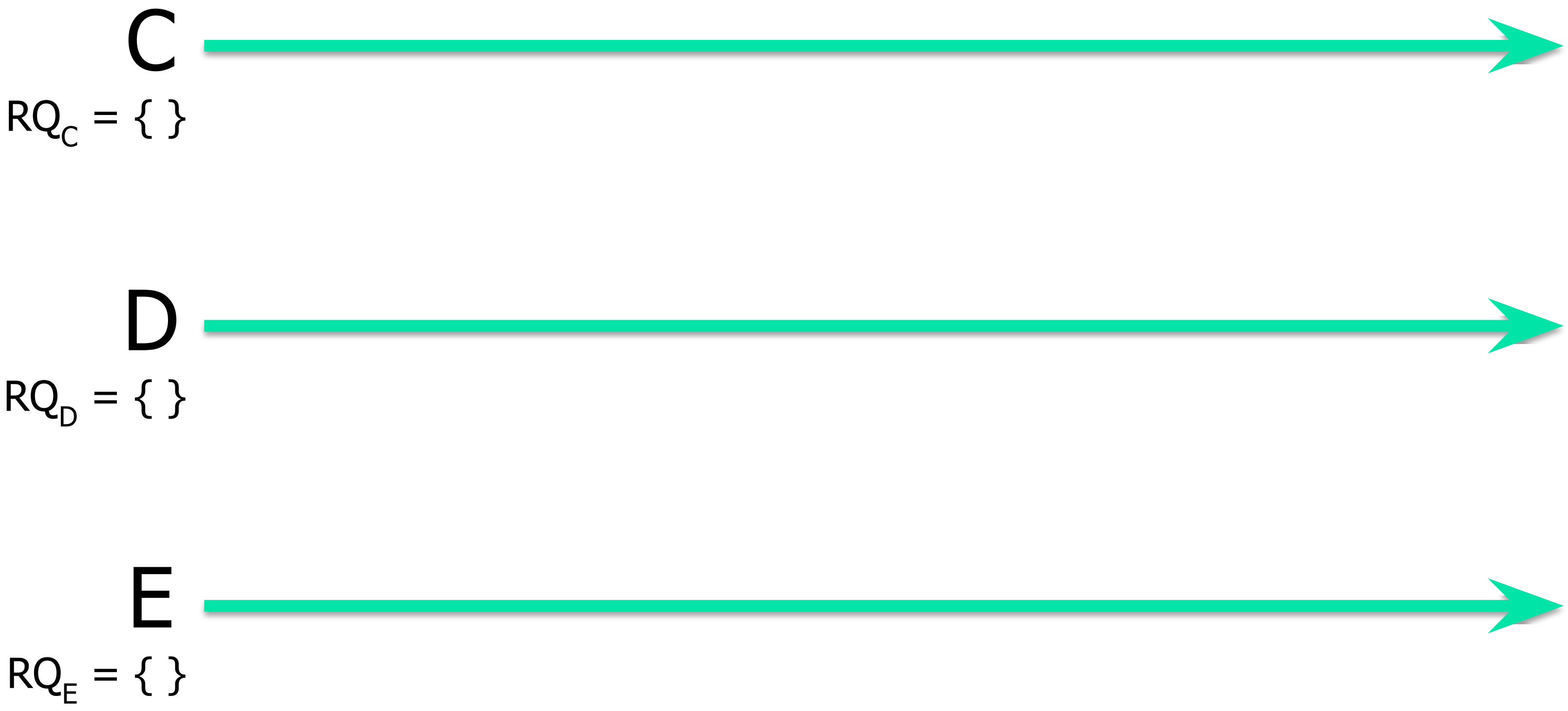
Lamport Algorithm

Rule 1

- To request a resource, process *i*
  - sends a request message T_m:*i* to every other process,
    - T_m is the timestamp of the message.
  - then stores a copy of the message in its request queue.

Lamport Example

- Total ordering: $C \prec D \prec E$



C

$RQ_C = \{ \}$

D

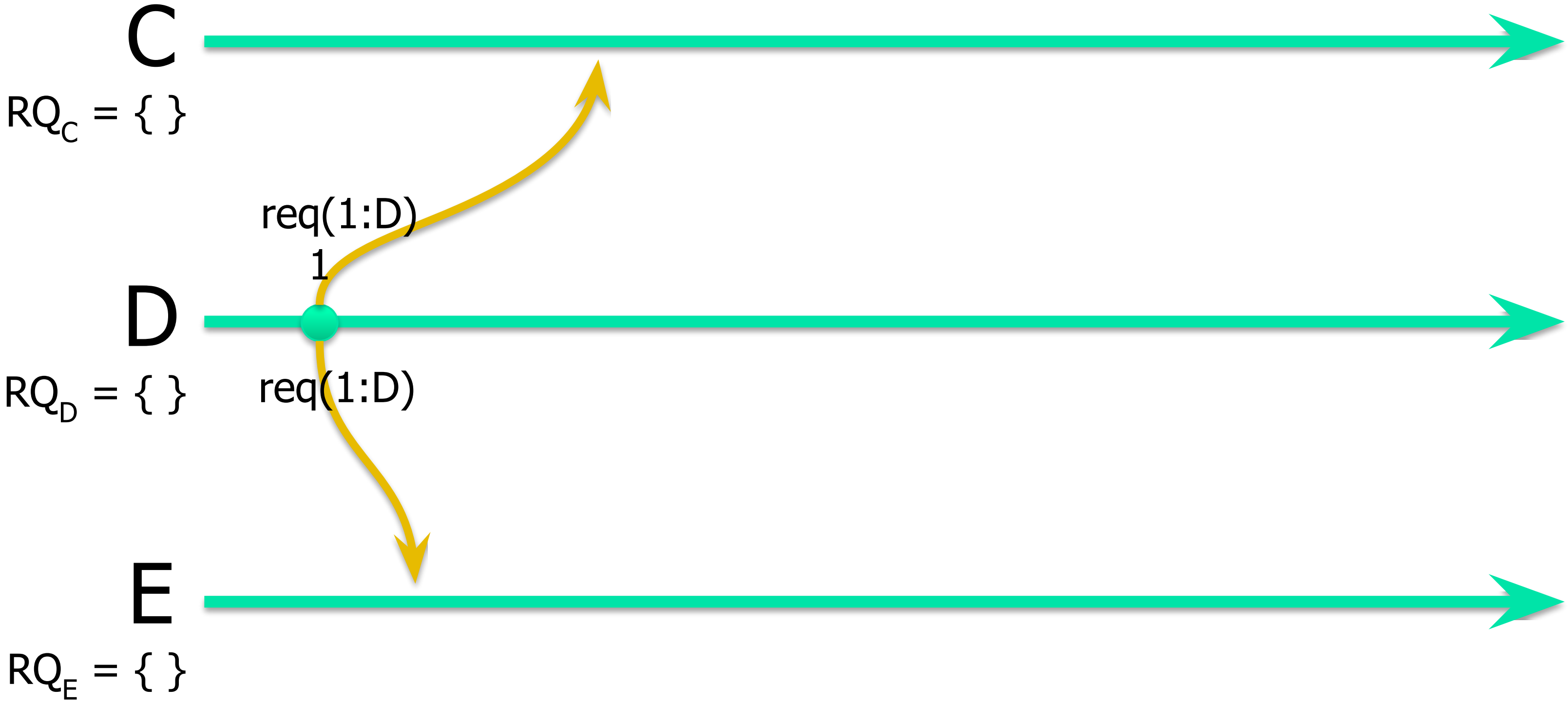
$RQ_D = \{ \}$

E

$RQ_E = \{ \}$

Lamport Example

- D requests the resource by sending request messages to C and E



C

$RQ_C = \{ \}$

req(1:D)

1

D

$RQ_D = \{ \}$

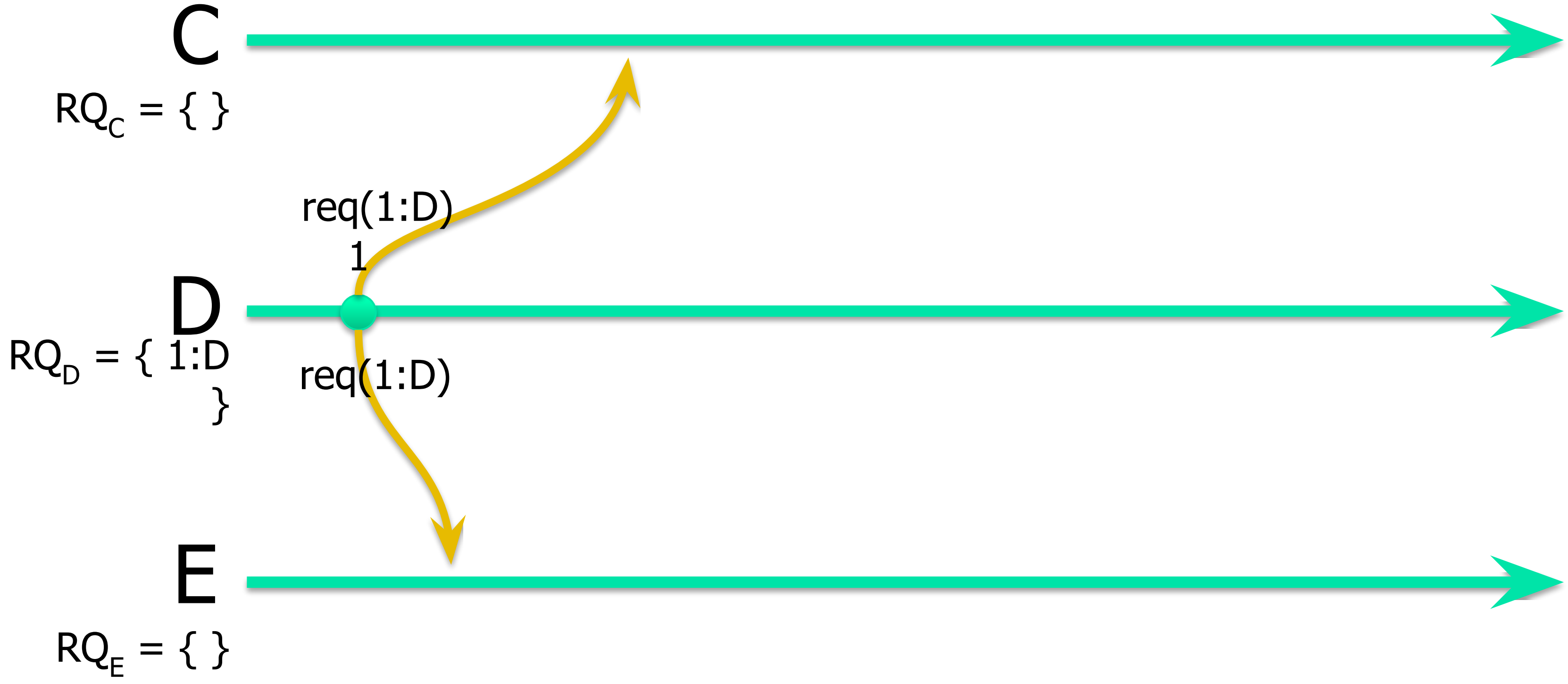
req(1:D)

E

$RQ_E = \{ \}$

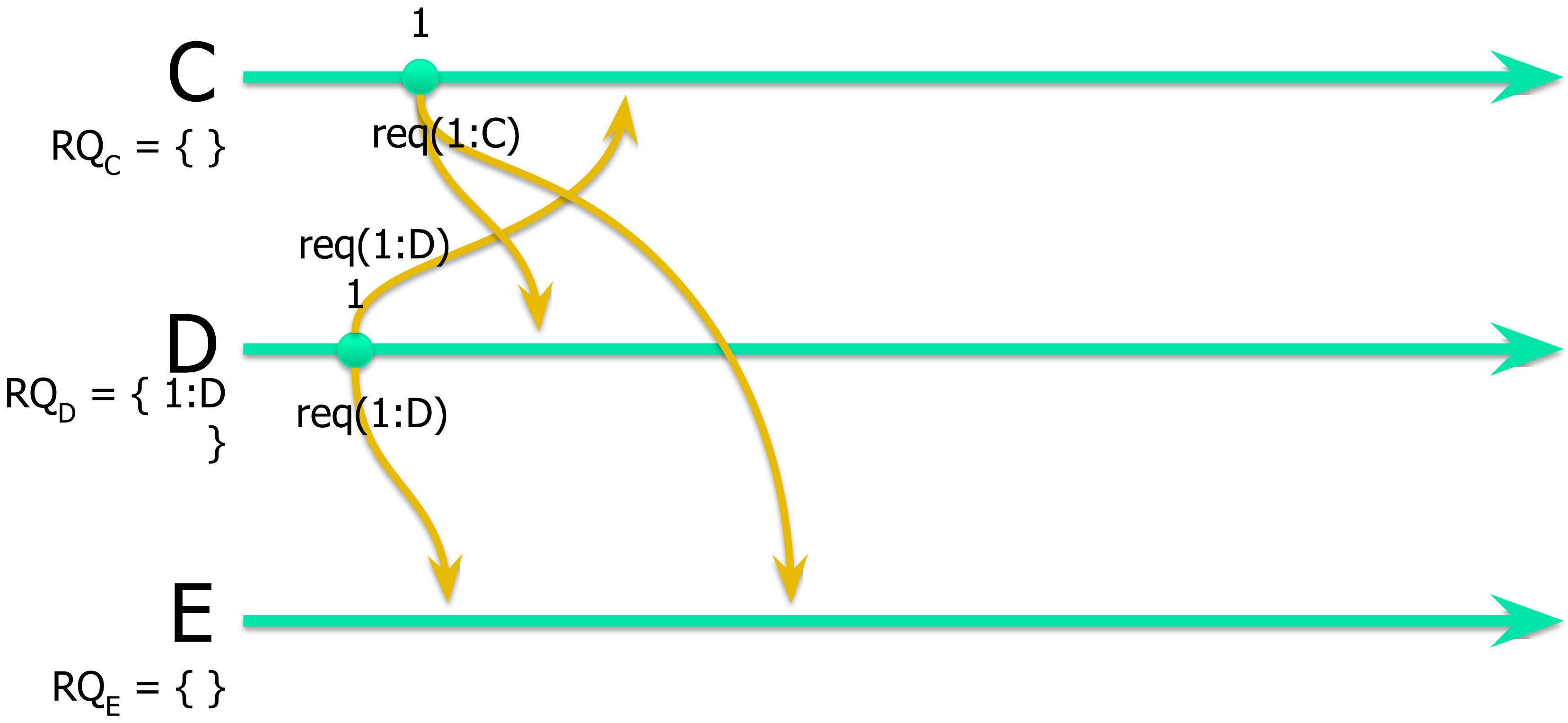
Lamport Example

- D then stores its own request



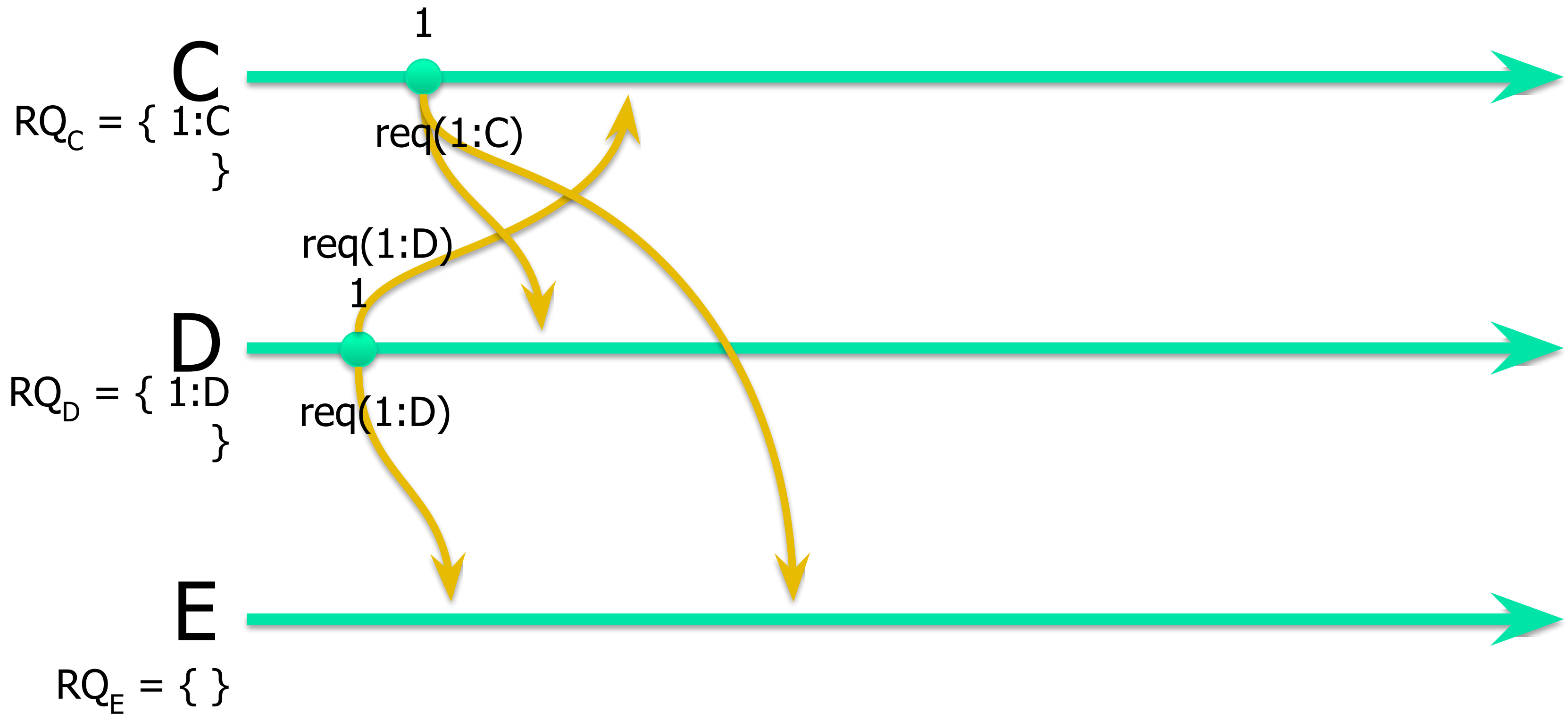
Lamport Example

- C requests the resource by sending request messages to D and E



Lamport Example

- C then stores its own request

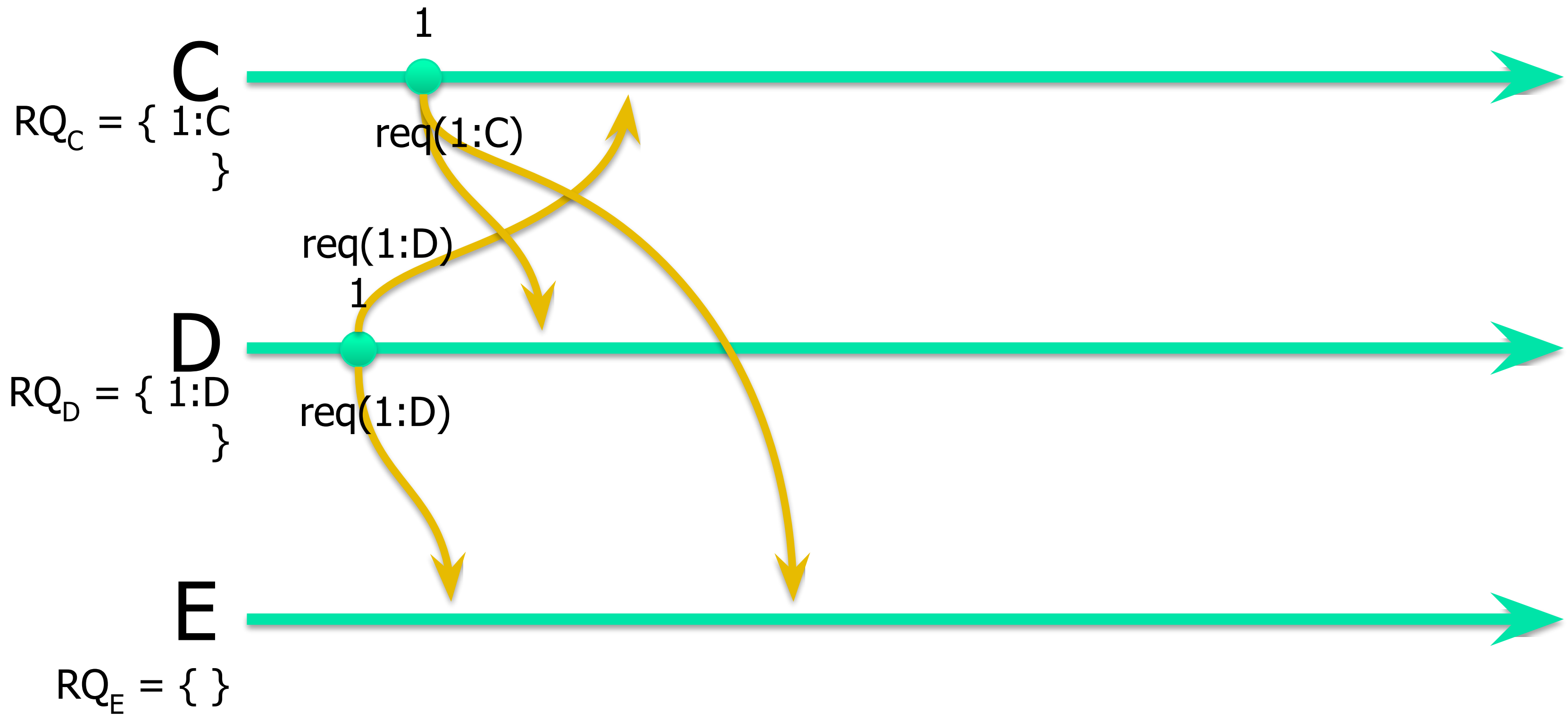


Lamport Algorithm

Rule 2

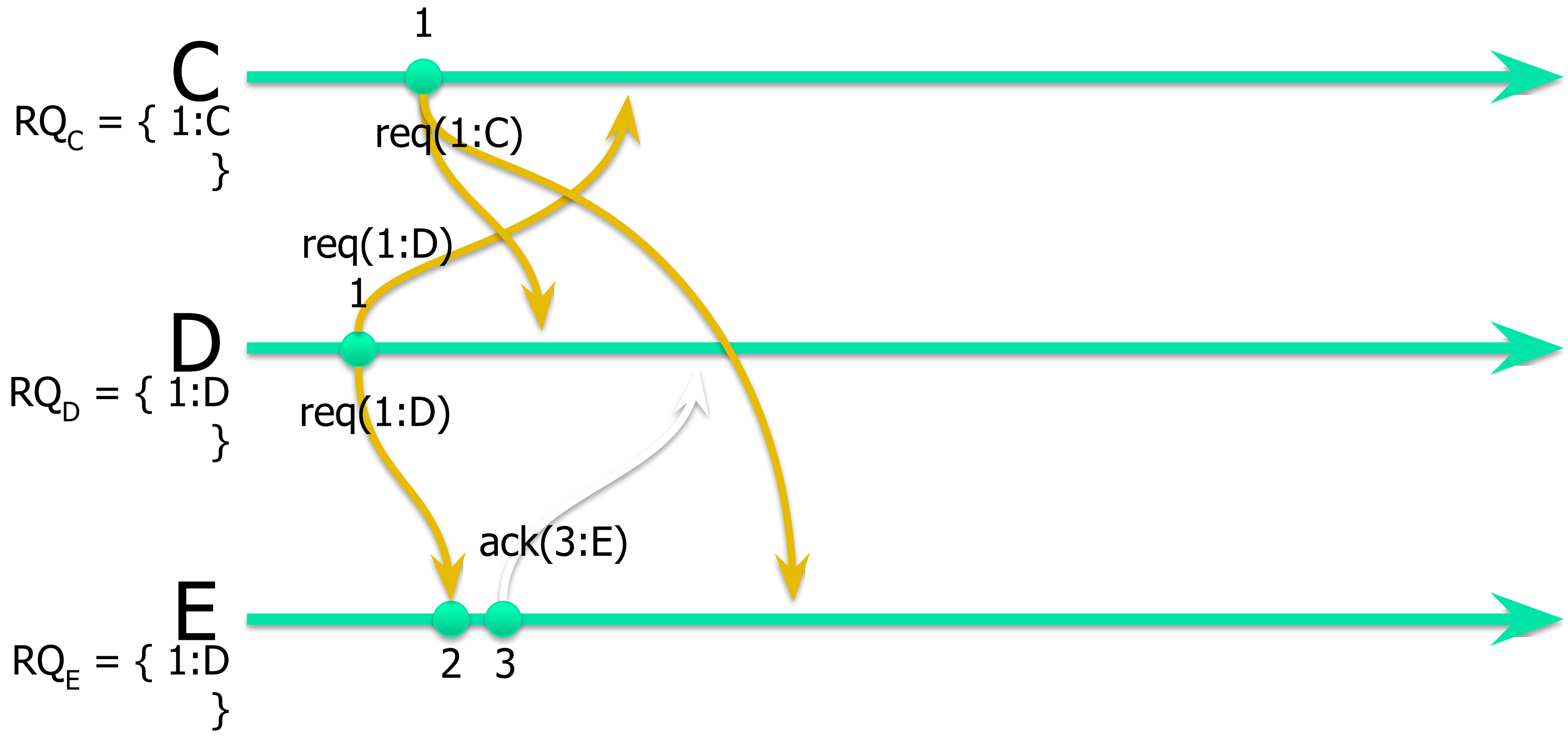
- When process *j* receives a request message T_m:*i*,
  - it places it in its request queue, and
  - sends a timestamped acknowledgment back to process *i*.

Lamport Example



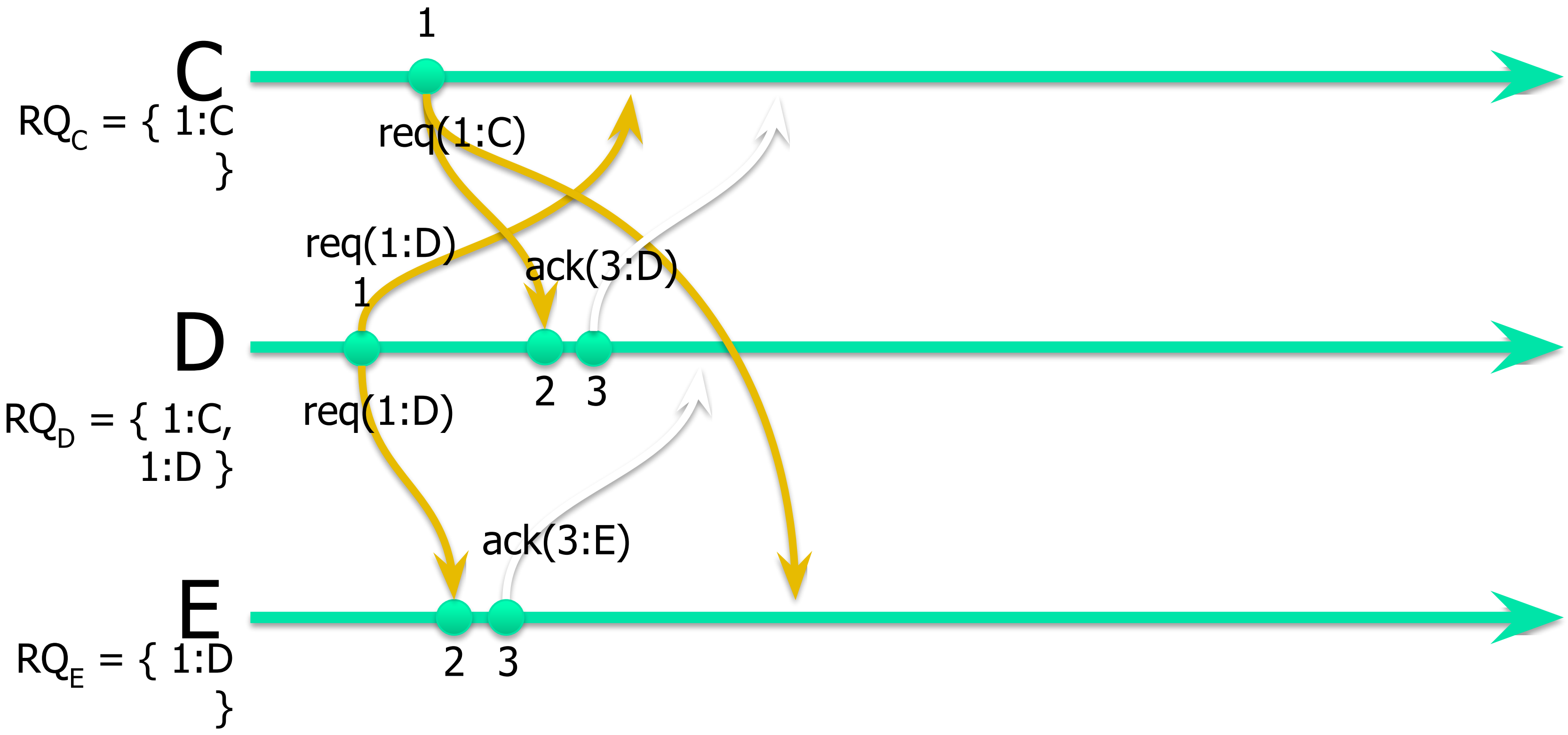
Lamport Example

- E places D’s request in its queue and sends an acknowledgment back



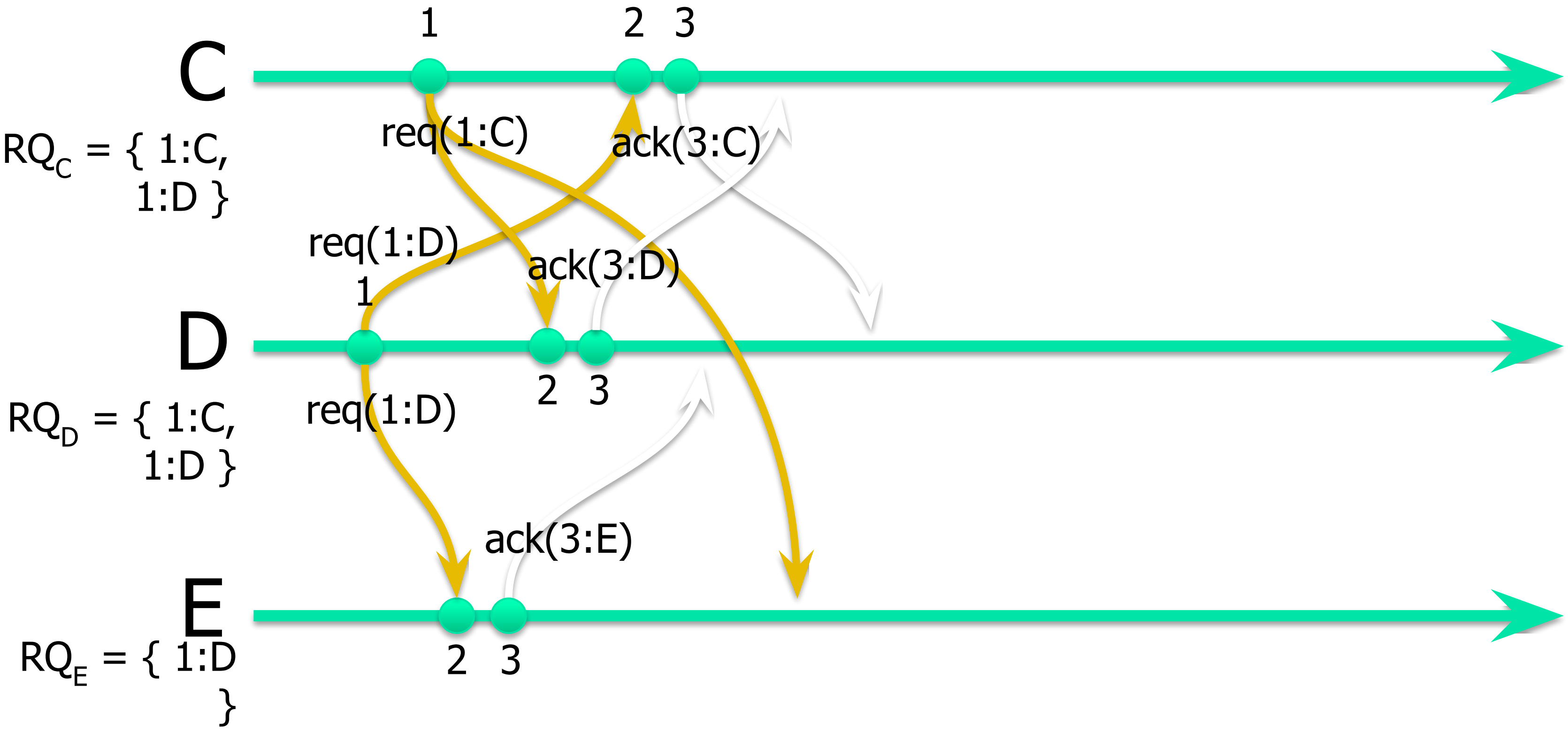
Lamport Example

- D places C’s request in its queue and sends an acknowledgment back



Lamport Example

- C places D's request in its queue and sends an acknowledgment back

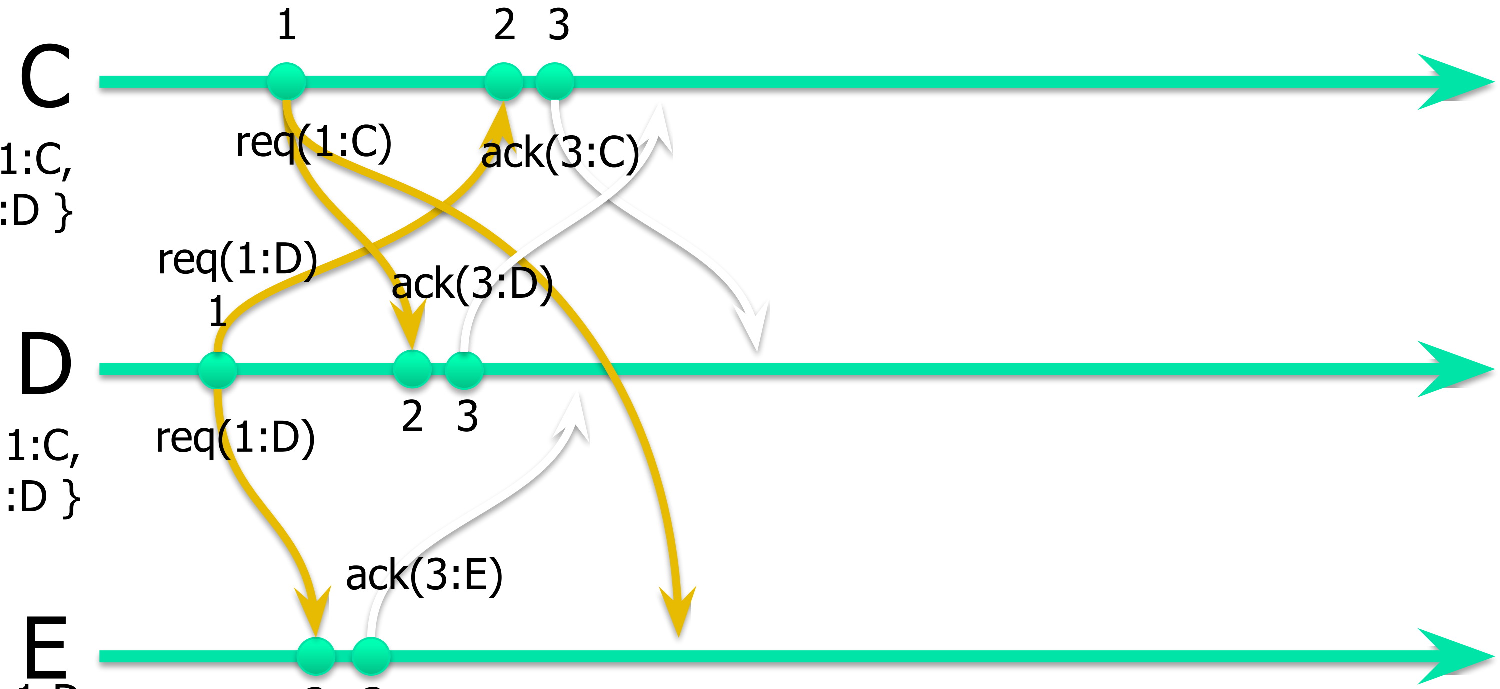


Lamport Algorithm

Rule 5

- Process *i* is granted the resource when the following conditions are satisfied:
  - There is a request T_m:*i* in its request queue that is ordered before any other request (by the total order)
  - Process *i* has received a message from every other process with a timestamp later than T_m

Lamport Example



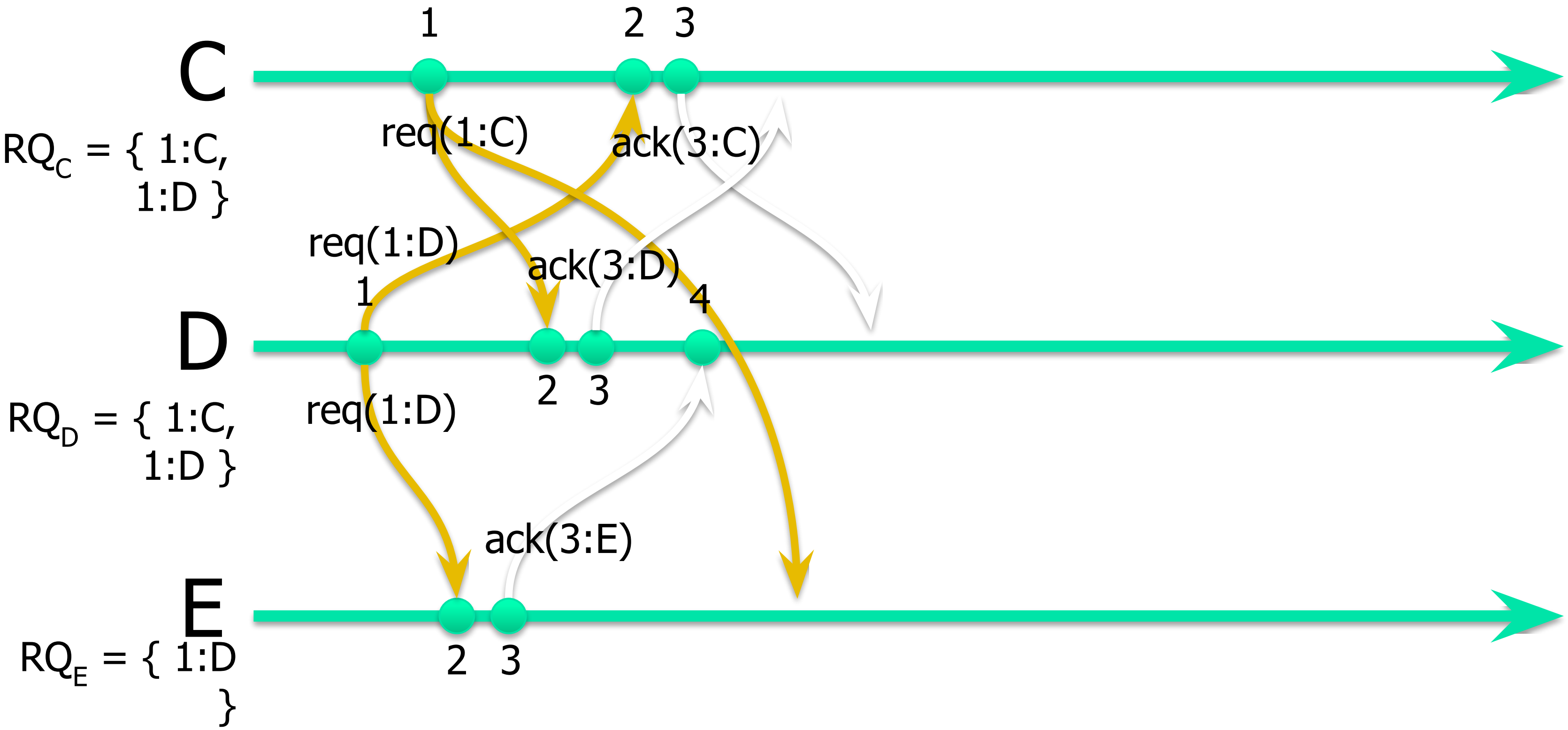
RQ_C = { 1:C,
1:D }

RQ_D = { 1:C,
1:D }

RQ_E = { 1:D
}

Lamport Example

- D isn’t granted the resource since 1:C ⇒ 1:D



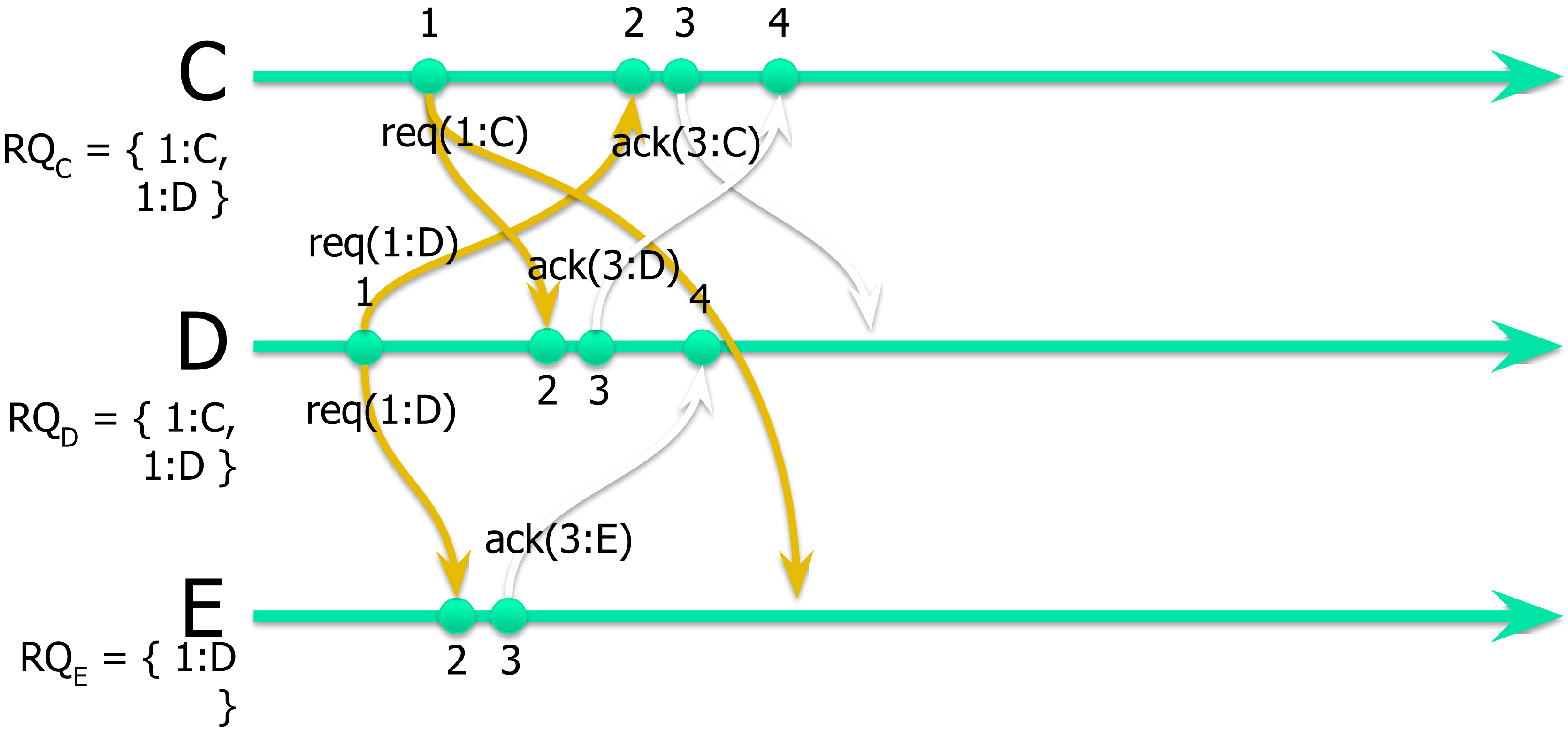
RQ_C = { 1:C, 1:D }

RQ_D = { 1:C, 1:D }

RQ_E = { 1:D }

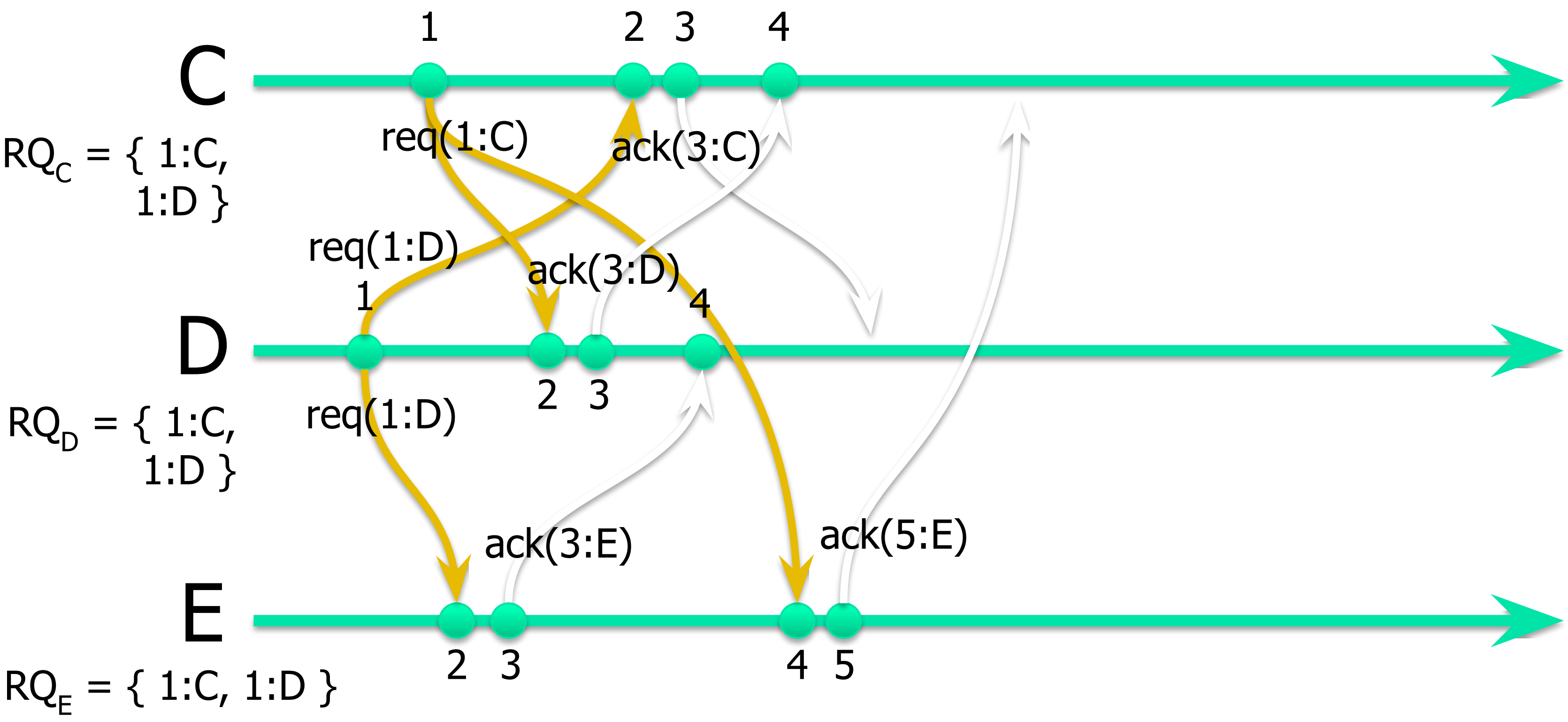
Lamport Example

- C isn't granted the resource since it has not received a message from E



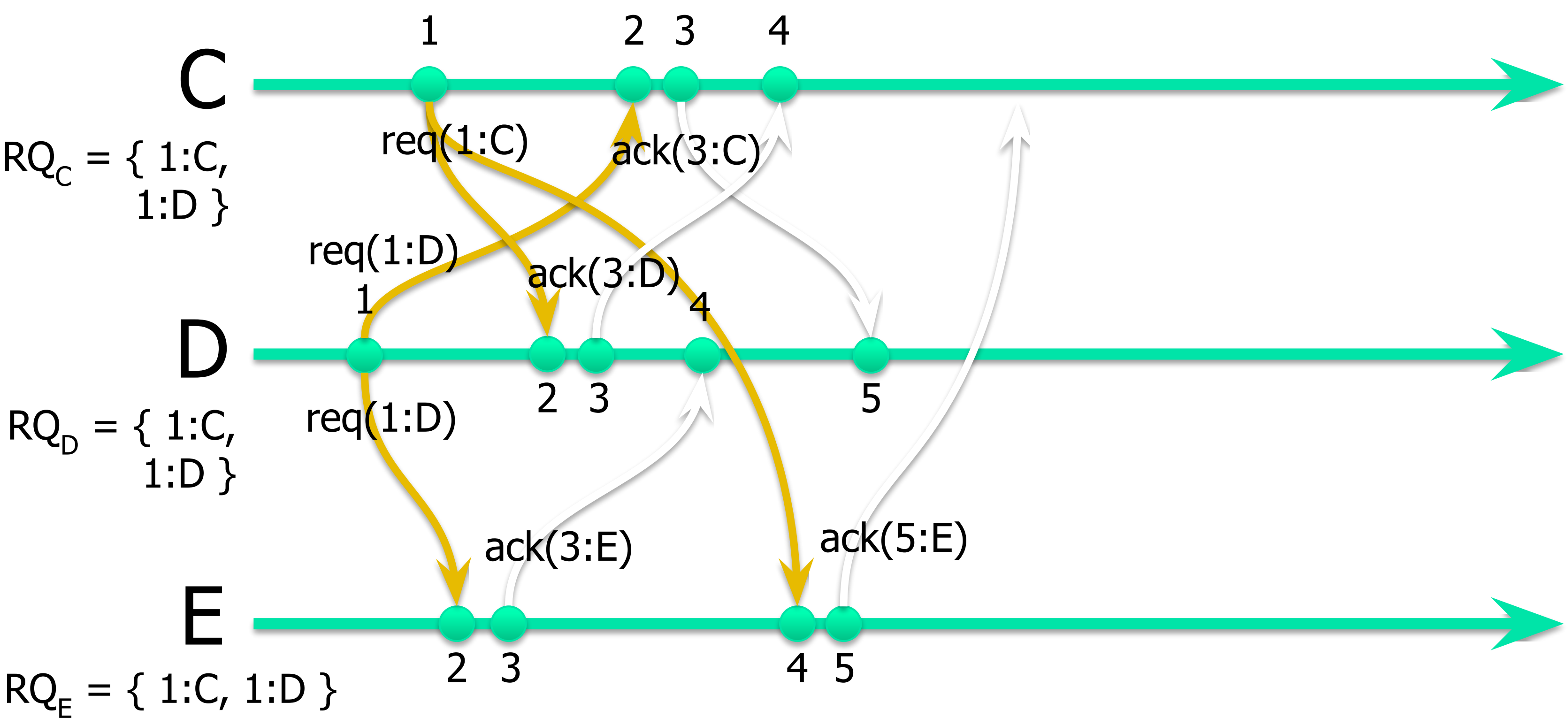
Lamport Example

- E places C’s request in its queue and sends an acknowledgment back



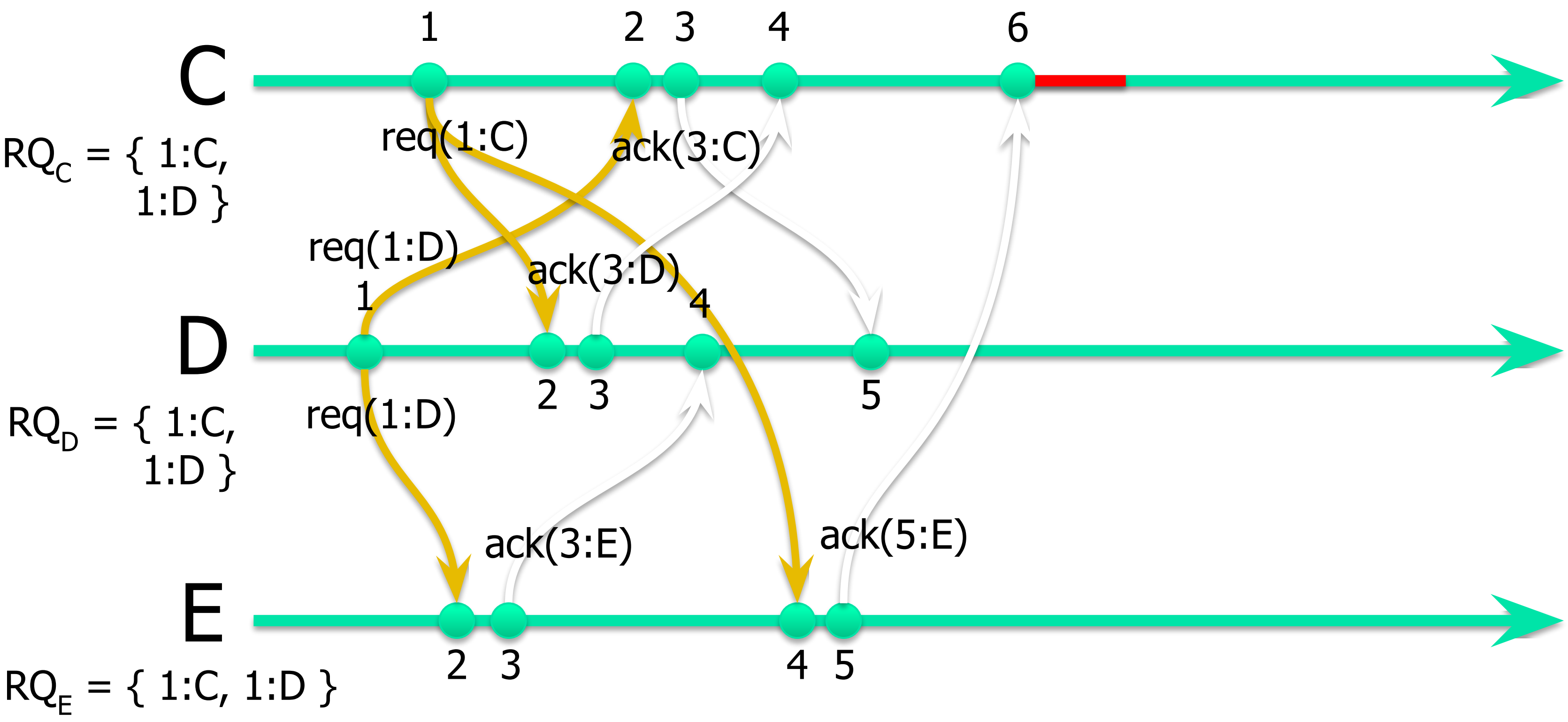
Lamport Example

- D isn't granted the resource since 1:C ⇒ 1:D



Lamport Example

- C is granted the resource since 1:C ⇒ 1:D and it has received later (i.e., timestamp > 1) messages from D and E



RQ_C = { 1:C, 1:D }

RQ_D = { 1:C, 1:D }

RQ_E = { 1:C, 1:D }

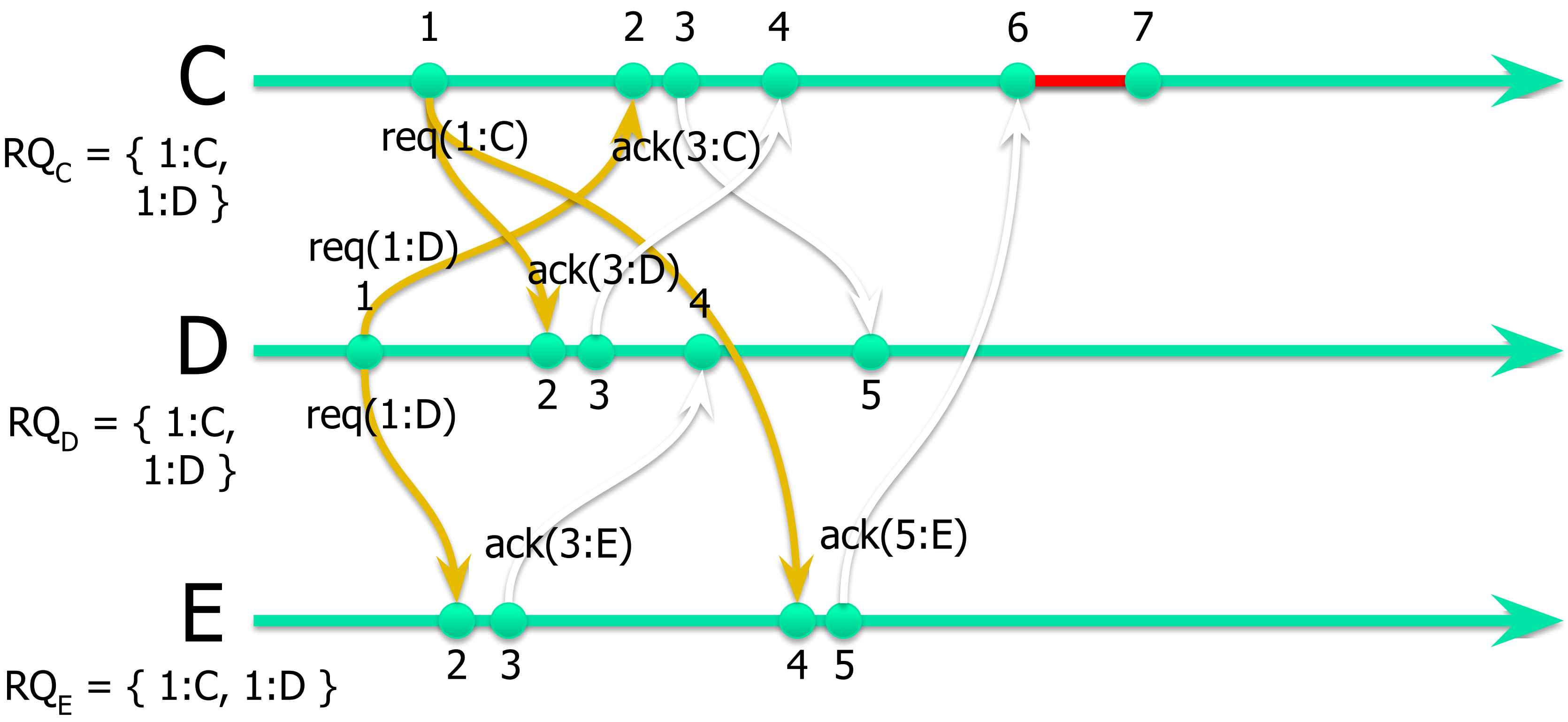
Lamport Algorithm

Rule 3

- To release a resource,
  - process *i* removes any T_m:*i* requests from its queue, and
  - sends a timestamped release message to every other process.

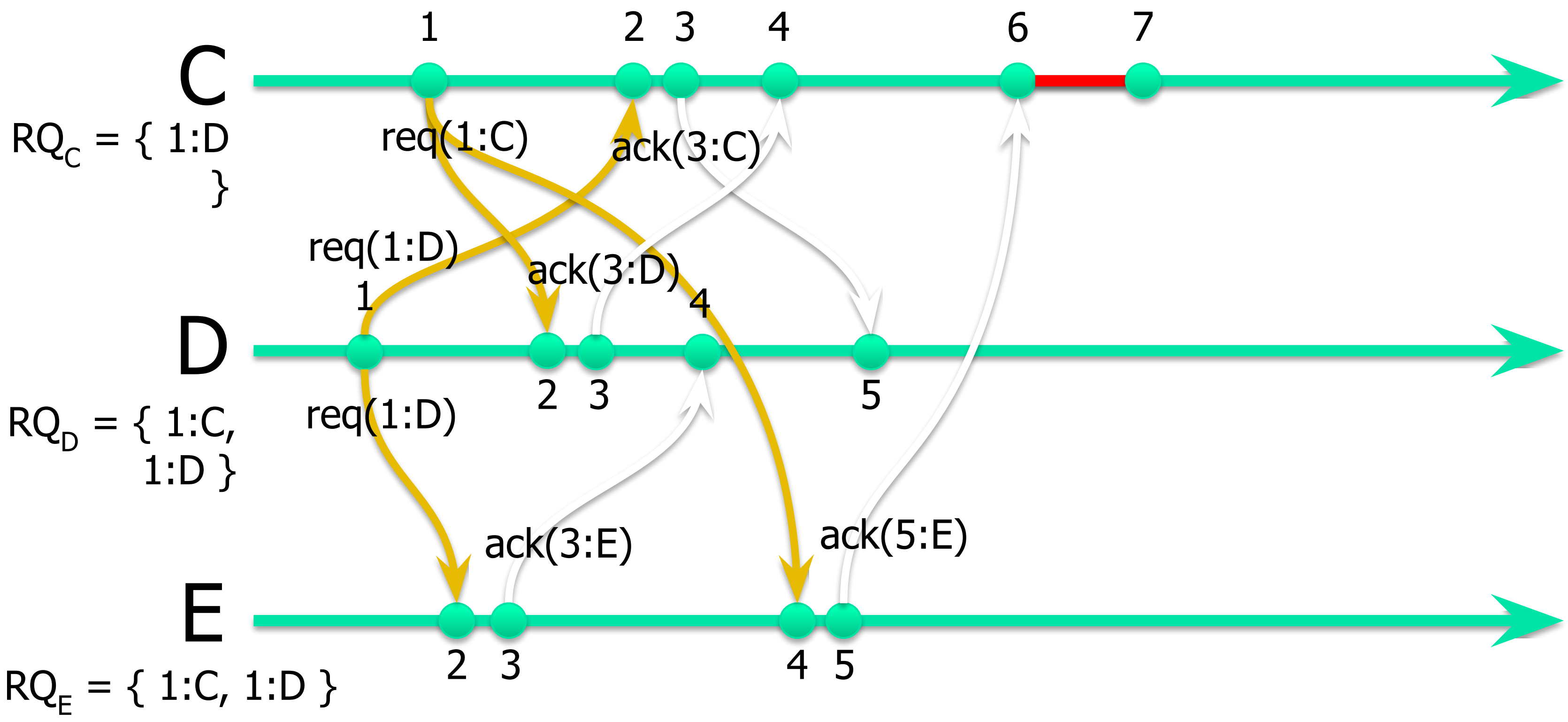
Lamport Example

- C is finished with resource



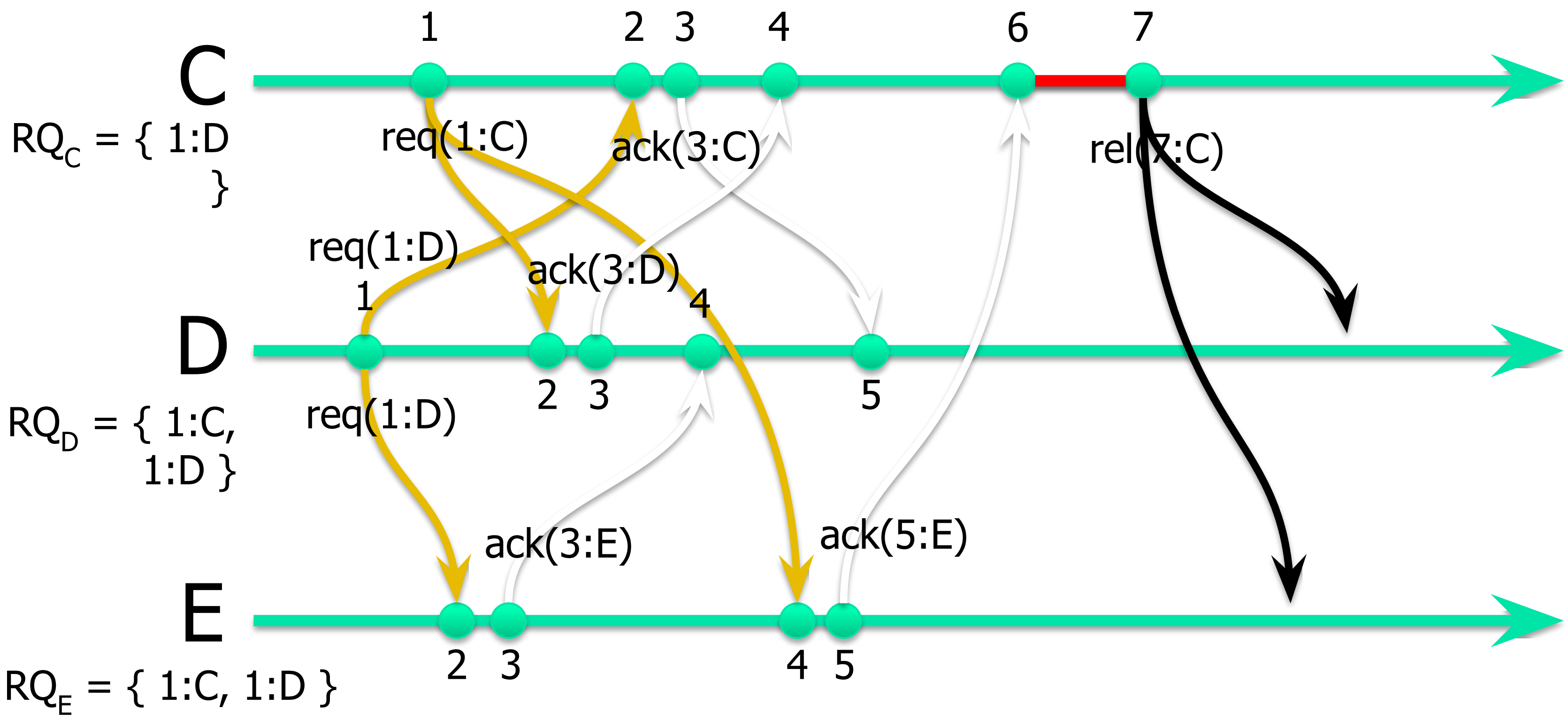
Lamport Example

- C removes its request from its own queue



Lamport Example

- C releases the resource by sending release messages to D and E

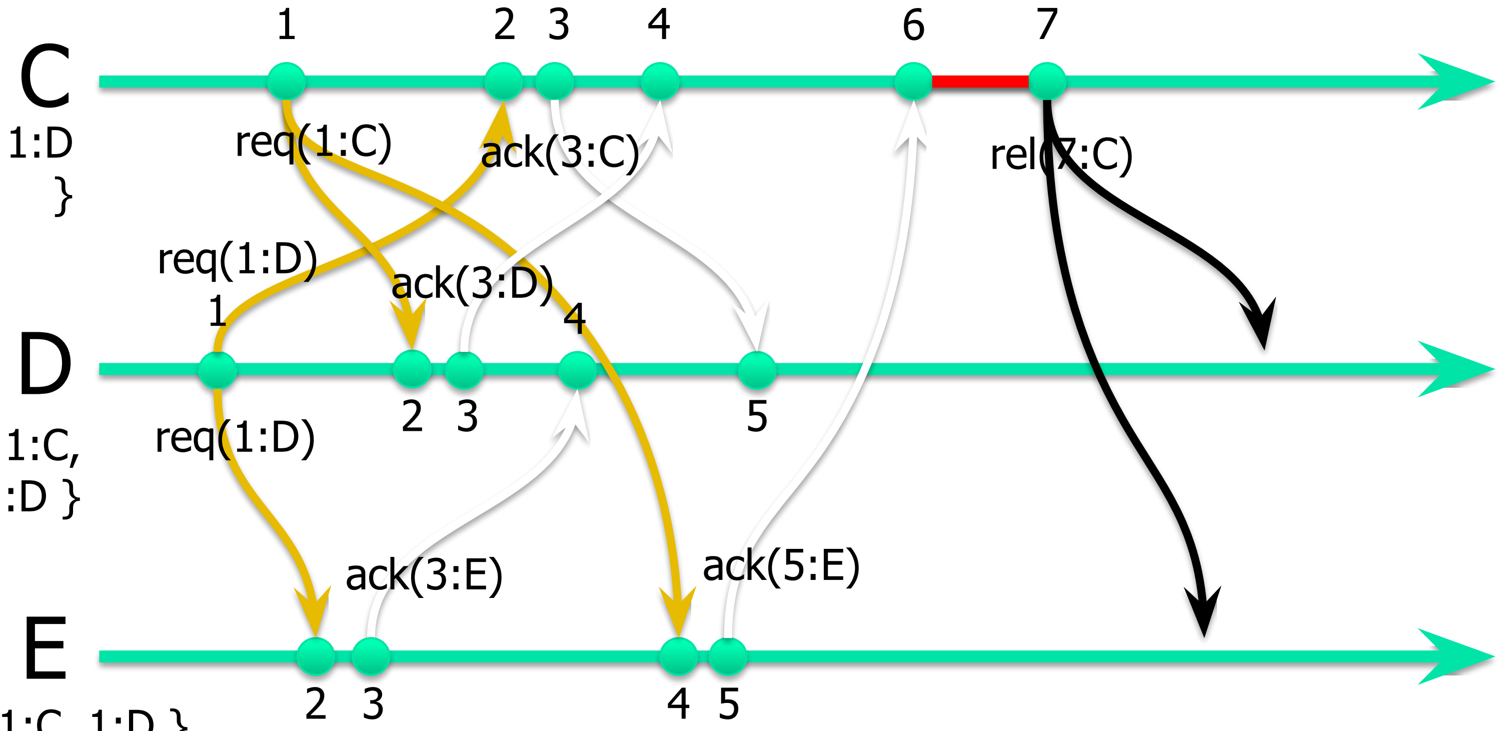


Lamport Algorithm

Rule 4

- When process *j* receives a release message from process *i*,
  - it removes any T_m:*i* requests from its request queue.

Lamport Example



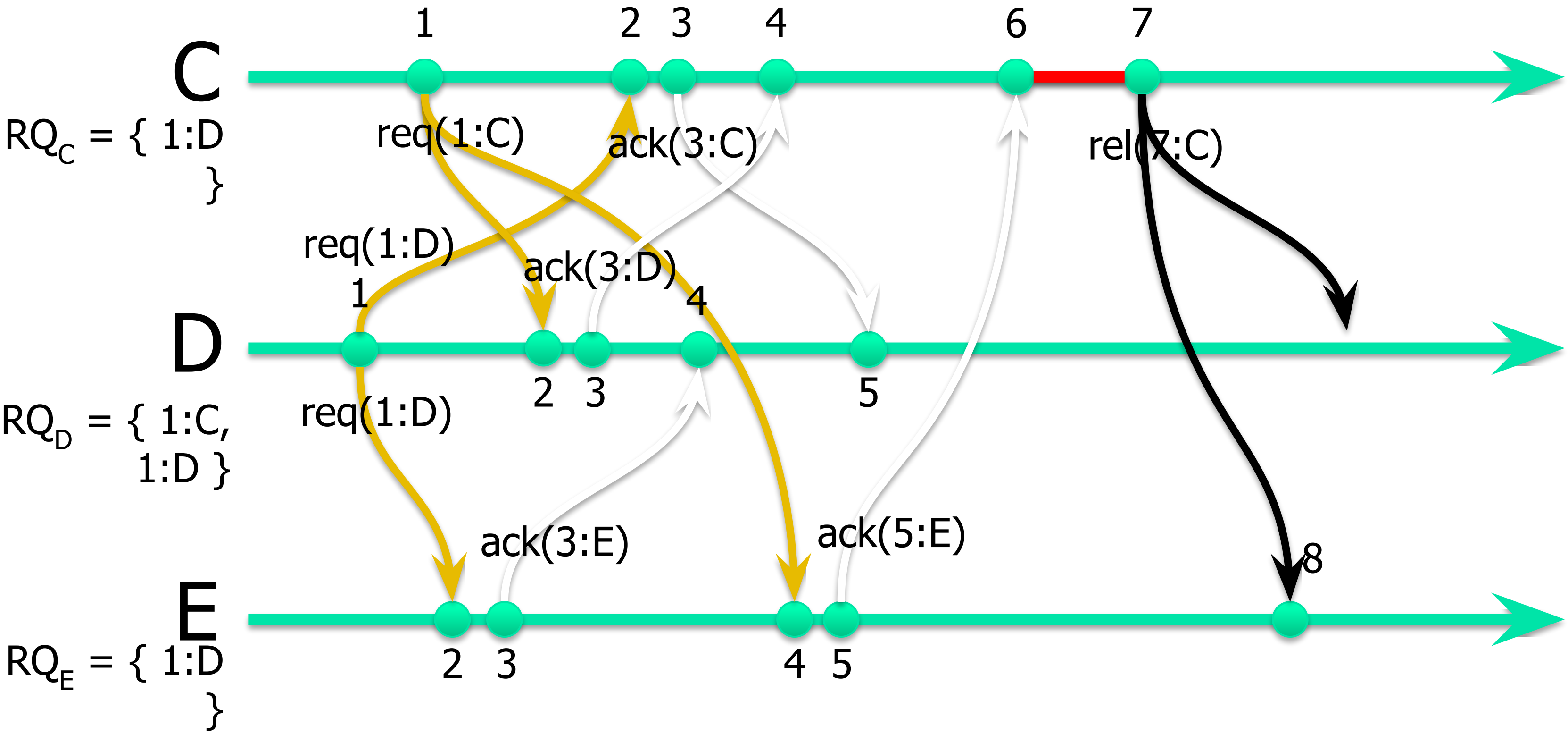
RQ_C = { 1:D
}

RQ_D = { 1:C,
1:D }

RQ_E = { 1:C, 1:D }

Lamport Example

- E removes C's request



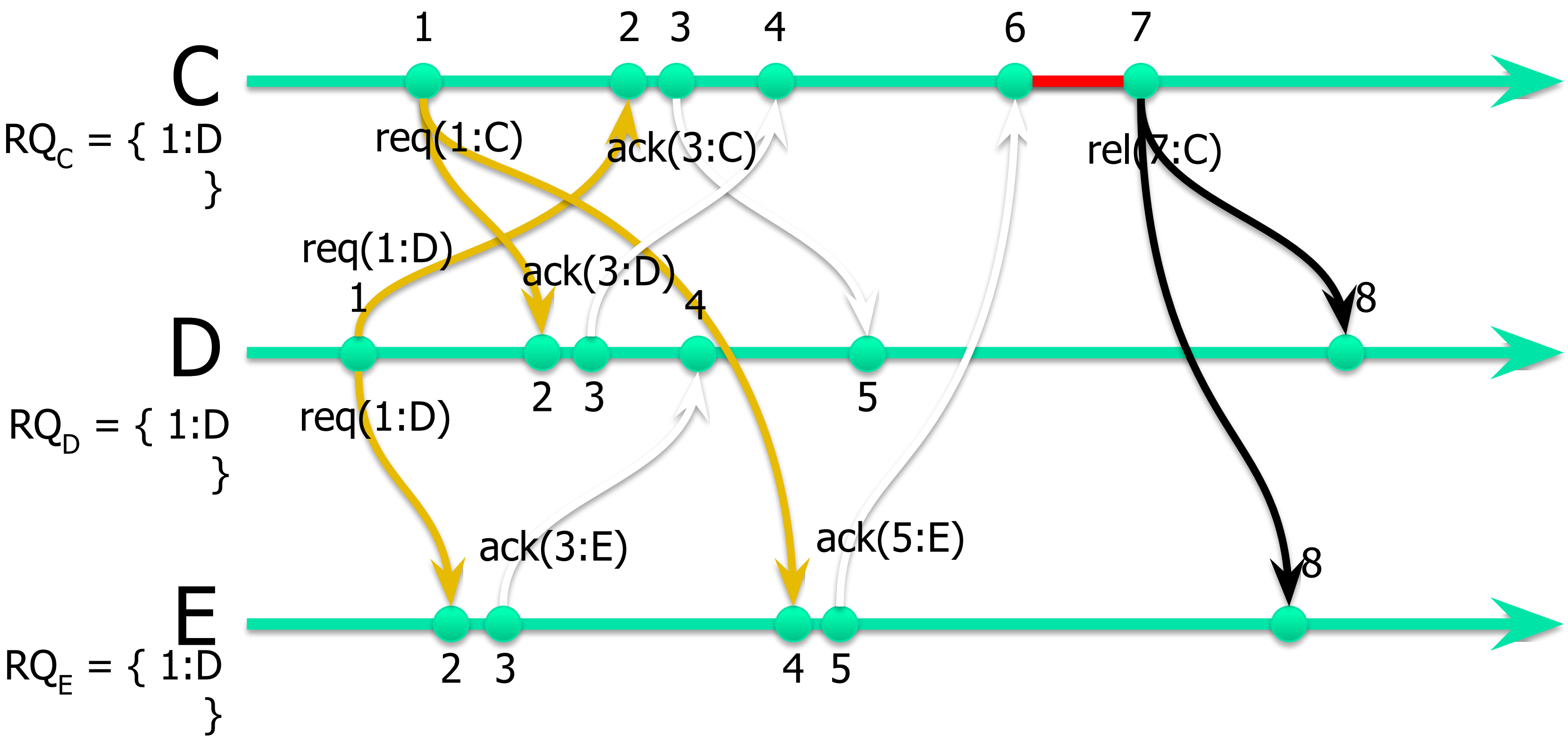
RQ_C = { 1:D }

RQ_D = { 1:C, 1:D }

RQ_E = { 1:D }

Lamport Example

- D removes C’s request



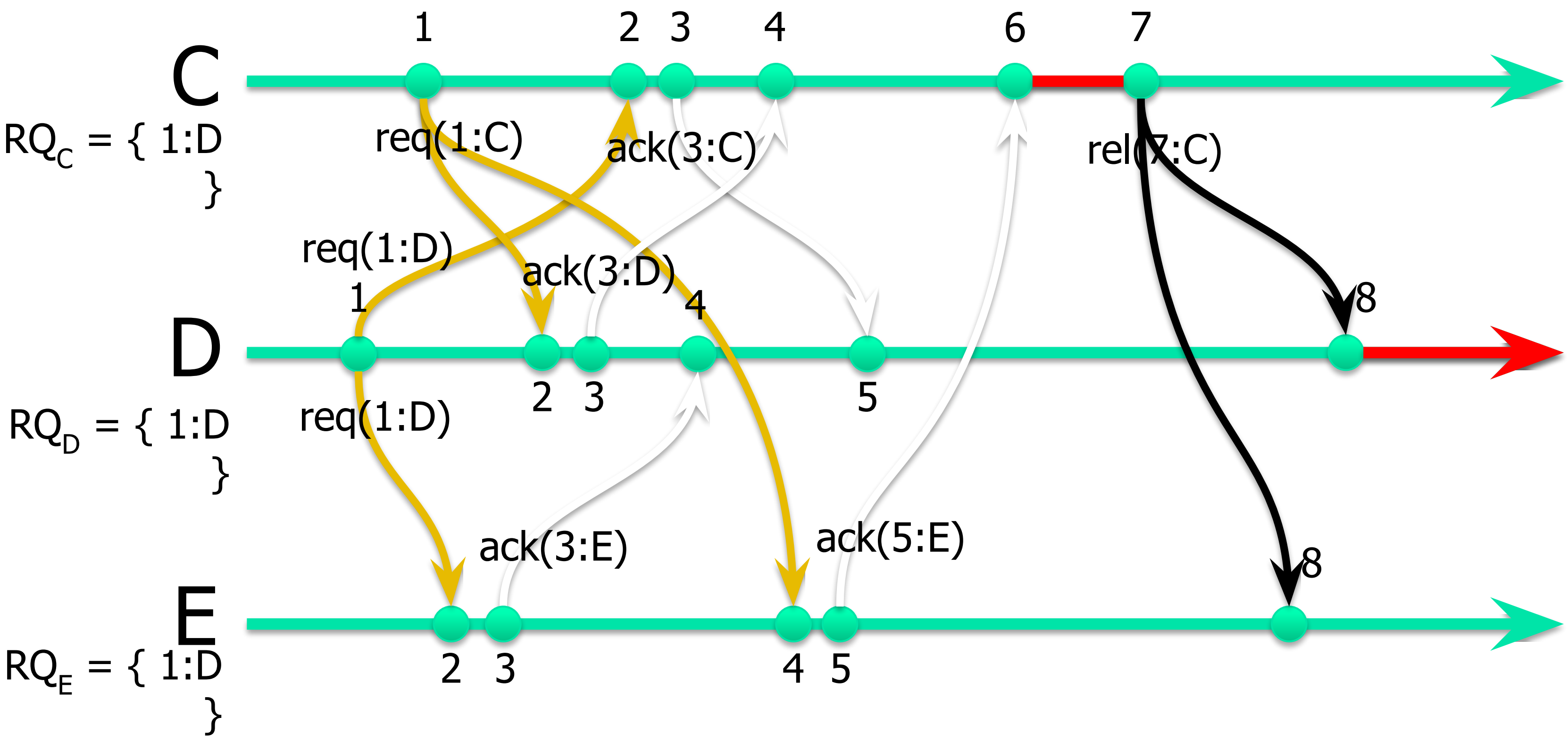
RQ_C = { 1:D }

RQ_D = { 1:D }

RQ_E = { 1:D }

Lamport Example

- D is granted the resource since it has received more later messages (timestamp > 1) from C and E



- Why is this cool?
  - Fair, short synchronization delay
  - Condition to enter critical section tested locally
  - Each process independently follows these rules,
    - there is no one coordinating process or central storage.
- Disadvantages:
  - Unreliable: any process failure halts progress

Other algorithms

- Ricart & Agrawal: optimizes Lamport to reduce the number of messages
  - Same drawbacks

- Voting based: more reliable – deals with failures / slow servers
  - But unfair (no ordering)

A voting-based protocol

Goal: Higher availability - deal with (temporary/ & full-stop) server failures, slow servers

Principle: Voting. The server protecting the resource is replicated *n* times, we'll call each server replica a coordinator

- Client access requires a majority vote from $m > n/2$ coordinators.
- A coordinator always responds immediately to a request.
  - (without contacting other coordinators)
- Client behavior if not enough votes: sends back votes and retries.

Failure model: If a coordinator crashes, it will (eventually) recover, but will have forgotten about the past (permissions it had granted).

Availability? Much better (load level dependednt?)

Correctness? Probabilistic! **Issue:** How robust is this system? What is the probability to make an incorrect 'grant acces' decision?

A voting-based protocol (cont)

Operating mode: Assume *n* coordinators (servers)

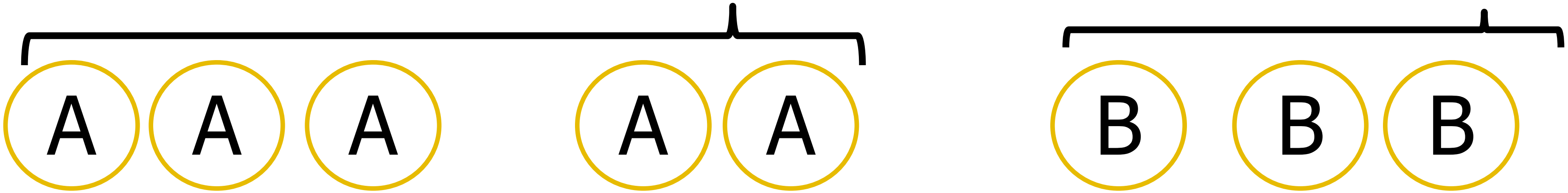
- A coordinator always responds immediately to a request.
- Access requires a majority vote from $m > n/2$ coordinators.
- If a coordinator fails, it restarts immediately but looses state

Issue: How robust is this system?

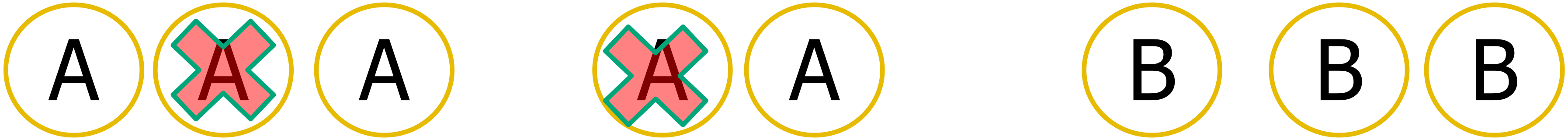
- *When may the system malfunction?*
- *How many servers have to 'flip'*
- *What's the chance?*

Malfunctioning?

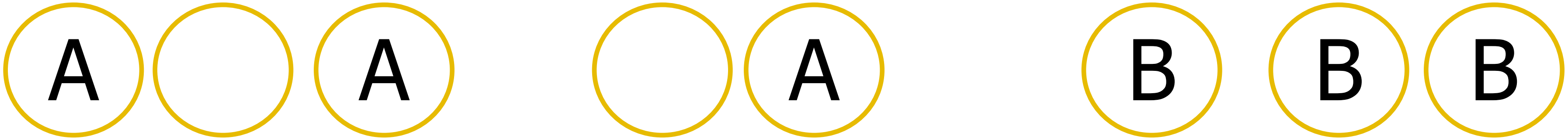
Initial state: decision made for A: M votes for A, N-M votes for B



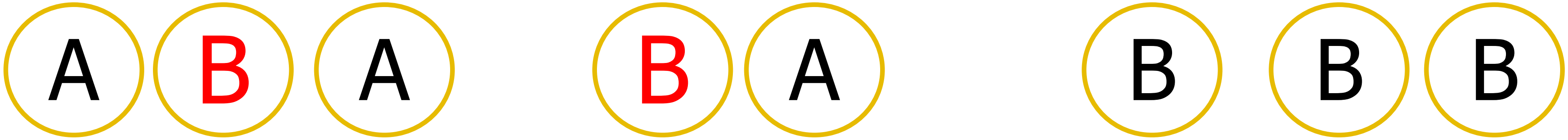
Some F nodes fail



And reset their state



Requests for B arrived later are effectively similar with a vote switch



Violation if (N-M) + F > M

A voting-based protocol (cont)

Operating mode: Assume *n* coordinators (servers)

- A coordinator always responds immediately to a request.
- Access requires a majority vote from $m > n/2$ coordinators.
- If a coordinator fails, it restarts immediately but looses state

Issue: How robust is this system?

What's the probability of malfunctioning

(violate mutual exclusion requirement and allow two clients in the critical region)?

- *p* the probability that a coordinator resets in the next *Δt* (crashes and recovers immediately)
    - *p = Δt / T, where* T is the an average server lifetime

- The probability that *k* out *m* coordinators reset during *Δt* $P[k]=C(k,m)p^{k}(1-p)^{m-k}$:

- Violation when at least *2m-n* coordinators reset

$$\sum_{k=2m-n}^{m} \binom{m}{k} p^{k}(1-p)^{m-k}$$

[conservative bound]

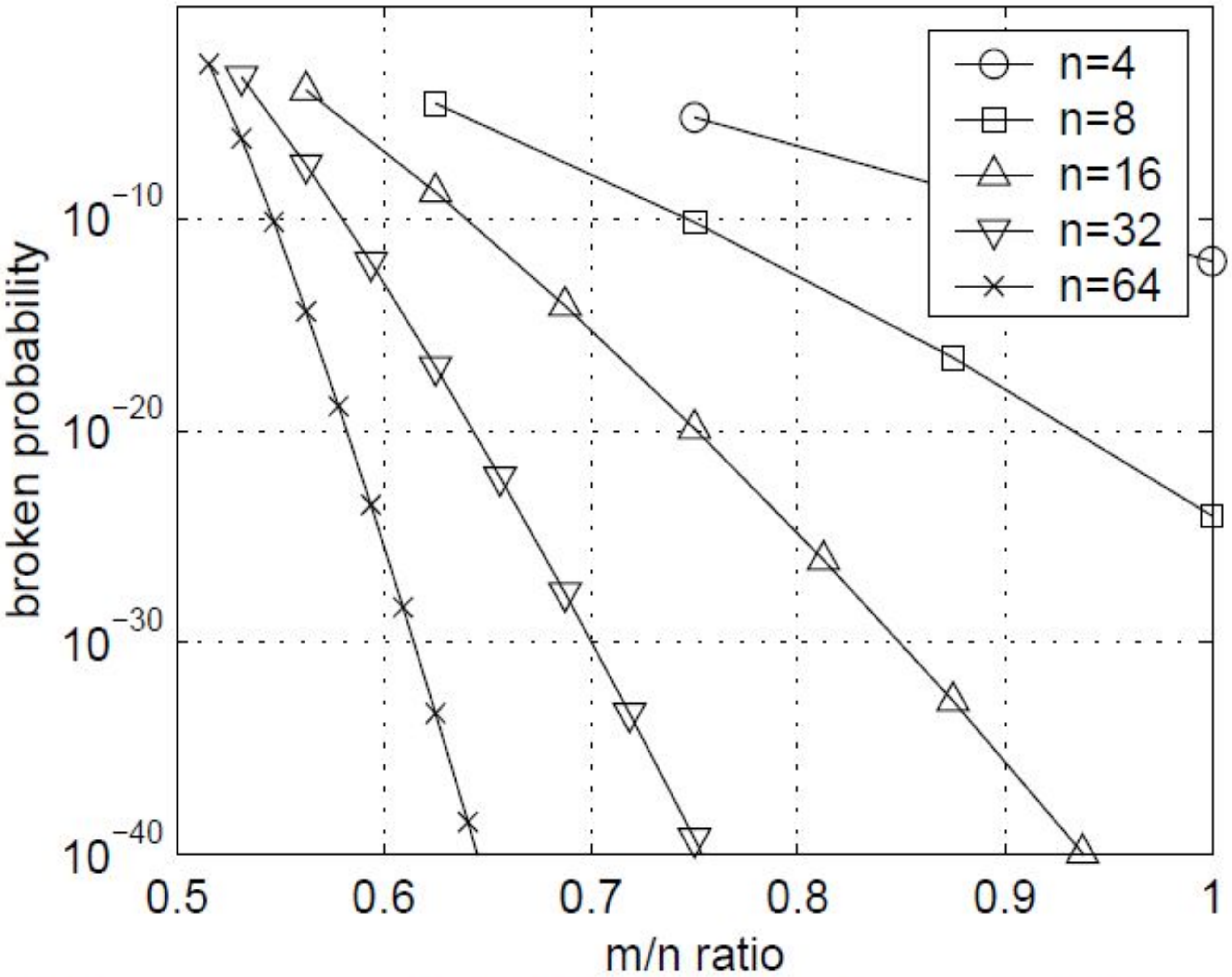


Figure 2. Probability to break exclusivity.

Issues

- Any possible issues with the design?
  - Deadlock?

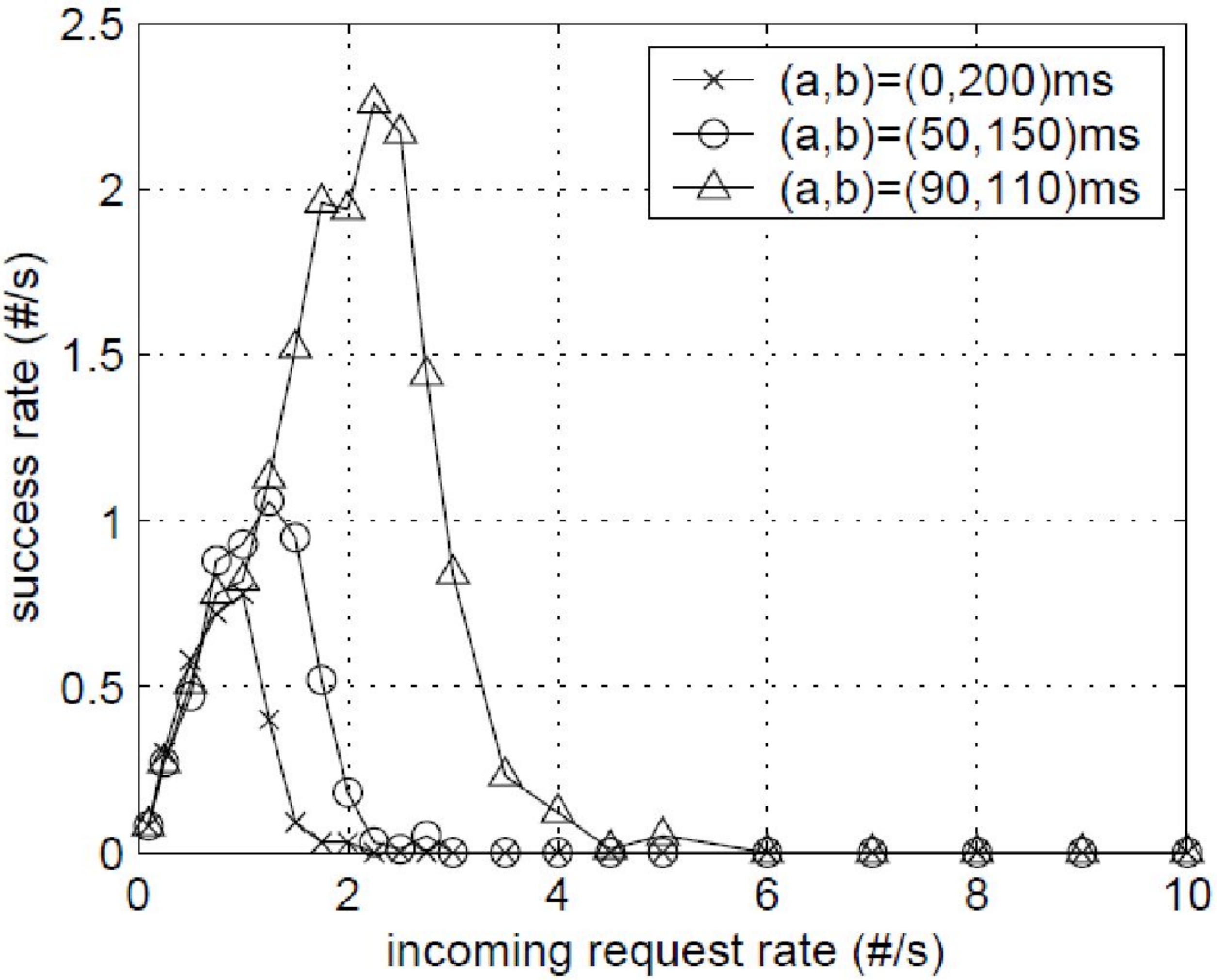
Recap: Assume *n* coordinators

- Access requires a majority vote from $m > n/2$ coordinators.
- A coordinator always responds immediately to a request.
- A client that does not gain a voting `round` sends back its votes.

How are we doing on various success criteria? fairness, avoid starvation, robustness (ability to deal with failures), low overhead, timeliness?

Performance issue – starvation

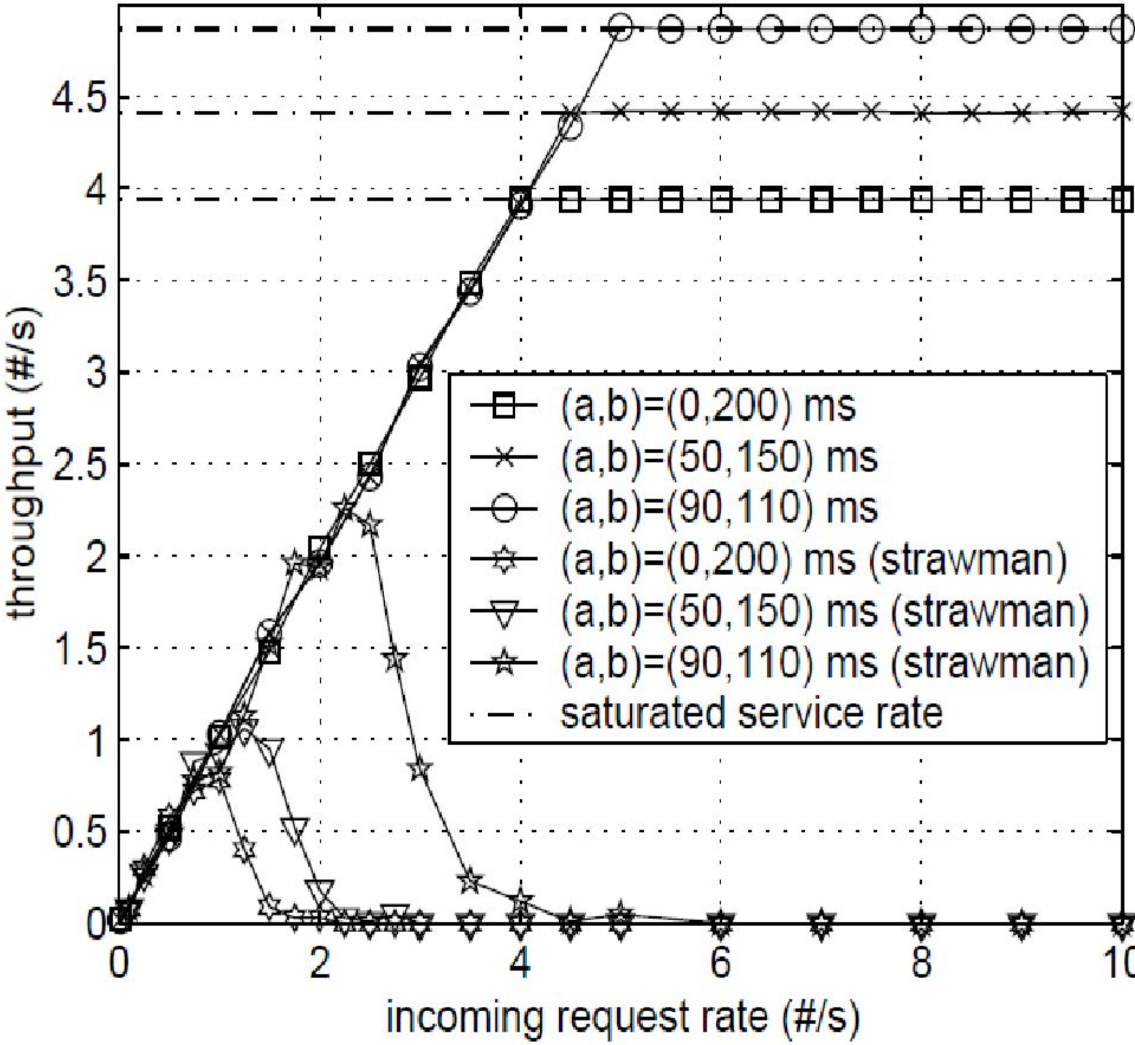
- Goodput with naïve solution



Performance issue – starvation

The fix:

- Exponential backoff +
- Informed by estimated place in the race



Algorithm Comparison

Algorithm	Messages per entry/exit	Synchronization delay (in RTTs)	Liveness
Central server	3	1 RTT	Bad: coordinator crash prevents progress
Token ring	N	<= sum(RTTs)/2	Horrible: any process' failure prevents progress
Lamport	3*(N-1)	max(RTT) across processes	Horrible: any process' failure prevents progress
Ricart & Agrawal	2*(N-1)	max(RTT) across processes	Horrible: any process' failure prevents progress
Voting	>= 2*(N-1) (might have vote recalls, too)	max(RTT) between the fastest N/2+1 processes	Great: can tolerate up to N/2-1 failures

You want the lock; no one else has it; how long till you get it?

So, Who Wins?

  - Well, none of the algorithms we’ve looked at thus far

But the closest one to industrial standards is…

- The centralized model (e.g., Google’s Chubby, Yahoo’s ZooKeeper)
- But replicate it for fault-tolerance across a few machines
  - Replicas coordinate via mechanisms similar to the ones we’ve shown for the distributed algorithms (e.g., voting) – we’ll talk later about generalized voting alg.
  - For manageable load, app writers must avoid using the centralized lock service as much as humanly possible!

Take-Aways

- Lamport algorithm
  - demonstrates how distributed processes can maintain consistent replicas of a data structure (the priority queue)
  - demonstrate utility of logical clocks
- Cost of a distributed system may be high.

\big_detour{end}