Dependable Distributed Systems
Master of Science in Engineering in Computer Science

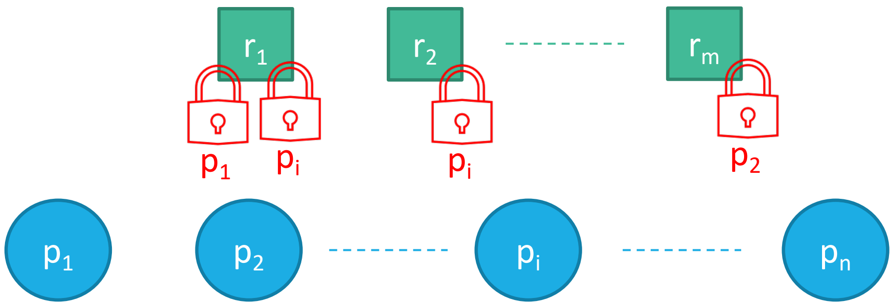
AA 2022/2023

LECTURE 5: DISTRIBUTED MUTUAL EXCLUSION

Recap - The Mutual Exclusion Problem

Let us consider

- a set of processes $\Pi = \{p_1, p_2, \ldots p_n\}$
- a set of resources $R = \{r_1, r_2, \ldots r_m\}$



PROBLEM

- Processes need to access resources exclusively and we need to design a distributed abstraction that allows them to coordinate to get access to resources

Recap - System Model

Let us consider

- a set of processes $\Pi = \{p_1, p_2, \ldots p_n\}$
- a set of resources $R = \{r_1, r_2, \ldots r_m\}$
  - For the sake of simplicity let us assume $|R| = 1$

The system is asynchronous

Processes are not going to fail (they will be always correct)

Processes communicate by exchanging messages on top of perfect point-to-point links

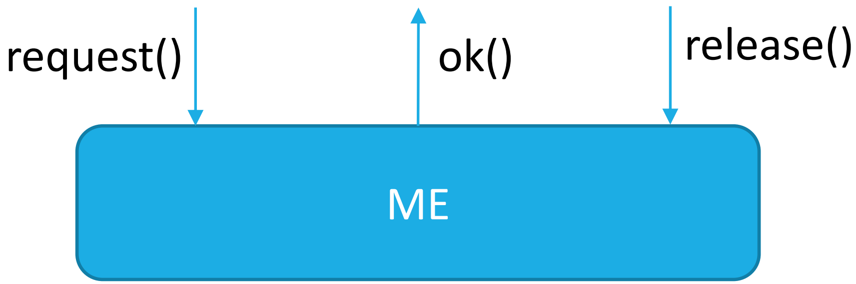
Recap - The Mutual Exclusion abstraction

EVENTS

- `request()`: it issues a request to enter into the critical section
- `ok()`: it notifies the process that it can now access the critical section
- `release()`: it is invoked to leave the critical section and to allow someone else to enter

PROPERTIES

- Mutual Exclusion: at any time t, at most one process *p* is running the critical section
- No-Deadlock: there always exists a process *p* able to enter the critical section
- No-Starvation: every `request()` and `release()` operation eventually terminate



Recap - Different Approaches to Distributed Mutual Exclusion

- Coordinator-based
- Token-based (over logical ring topology)
- Quorum-based
- Logical Clock-based

Coordinator-based Distributed Mutual Exclusion

BASIC IDEA

- There exist a special process (i.e., a coordinator) that collects requests and grant permission to enter into the critical section

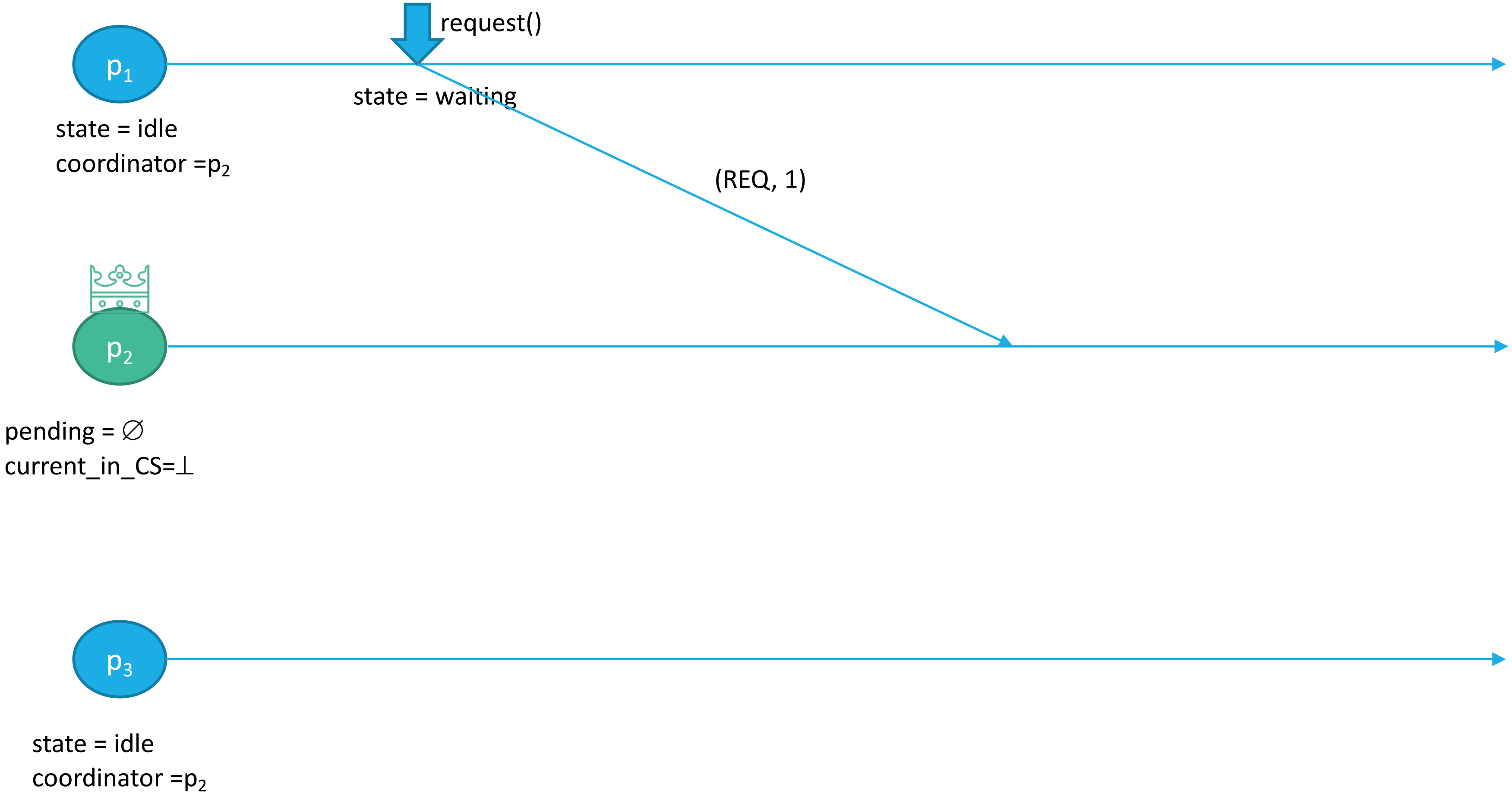
```
Init
state = idle
coordinator = getCoordinatorId()

upon event request()
    state = waiting
    trigger pp2pSend(REQ, i) to coordinator

upon event pp2pDeliver(GRANT_CS)
    state = CS
    trigger ok()

upon event release()
    state = idle
    trigger pp2pSend(REL, i) to coordinator
```

PROCESS CODE

```
Init
pending = ∅
current_in_CS=⊥

upon event pp2pDeliver(REQ, j) from p_j
        pending = pending ∪ {p_j}

when pending ≠ ∅ and current_in_CS=⊥
        candidate=select_process (pending)
        pending = pending \ candidate
        current_in_CS = candidate
        trigger pp2pSend(GRANT_CS) to candidate

upon event pp2pDeliver(REL, j) from p_j
        if current_in_CS = j
                current_in_CS = ⊥
```

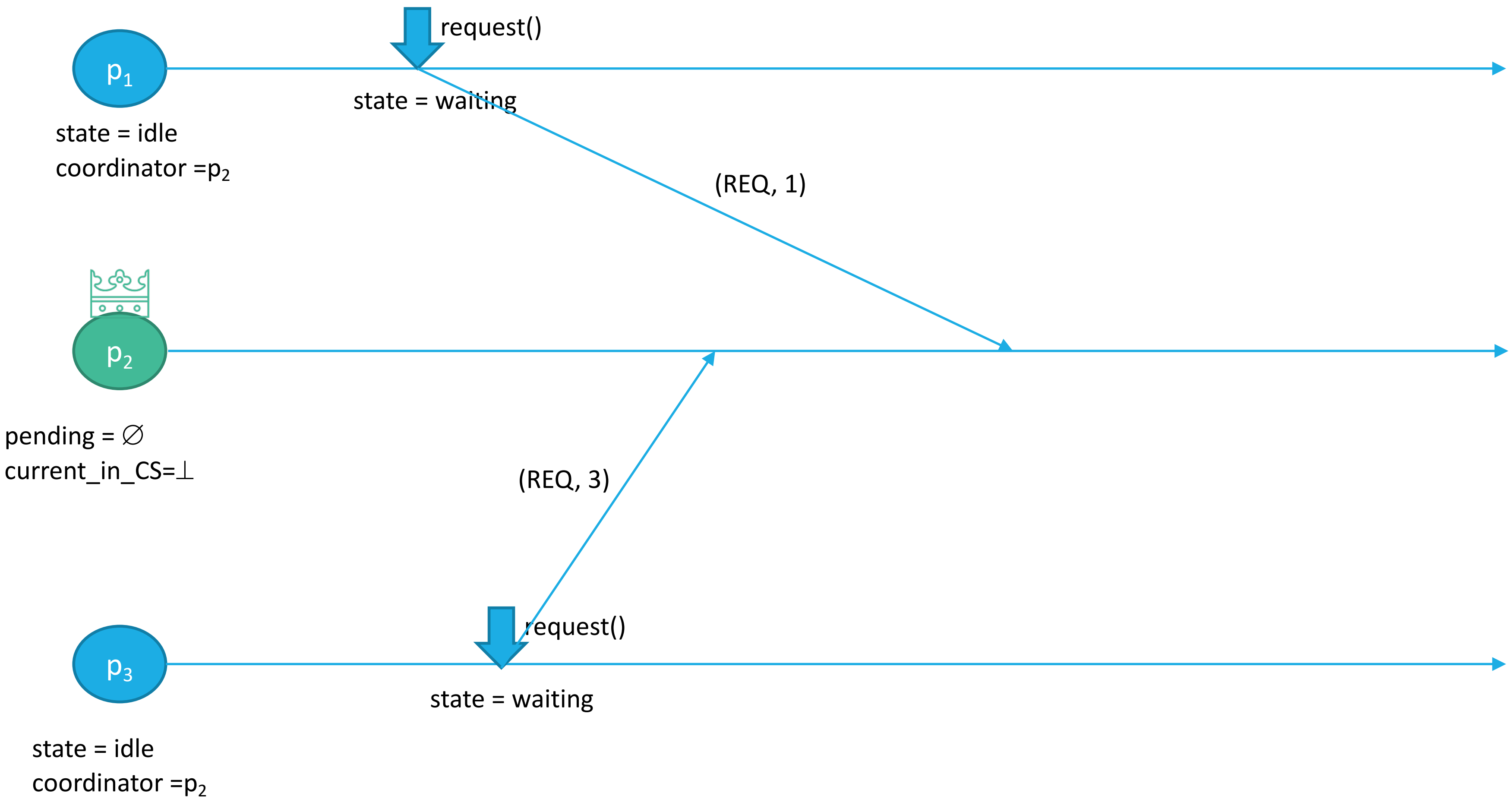
COORDINATOR CODE

Example

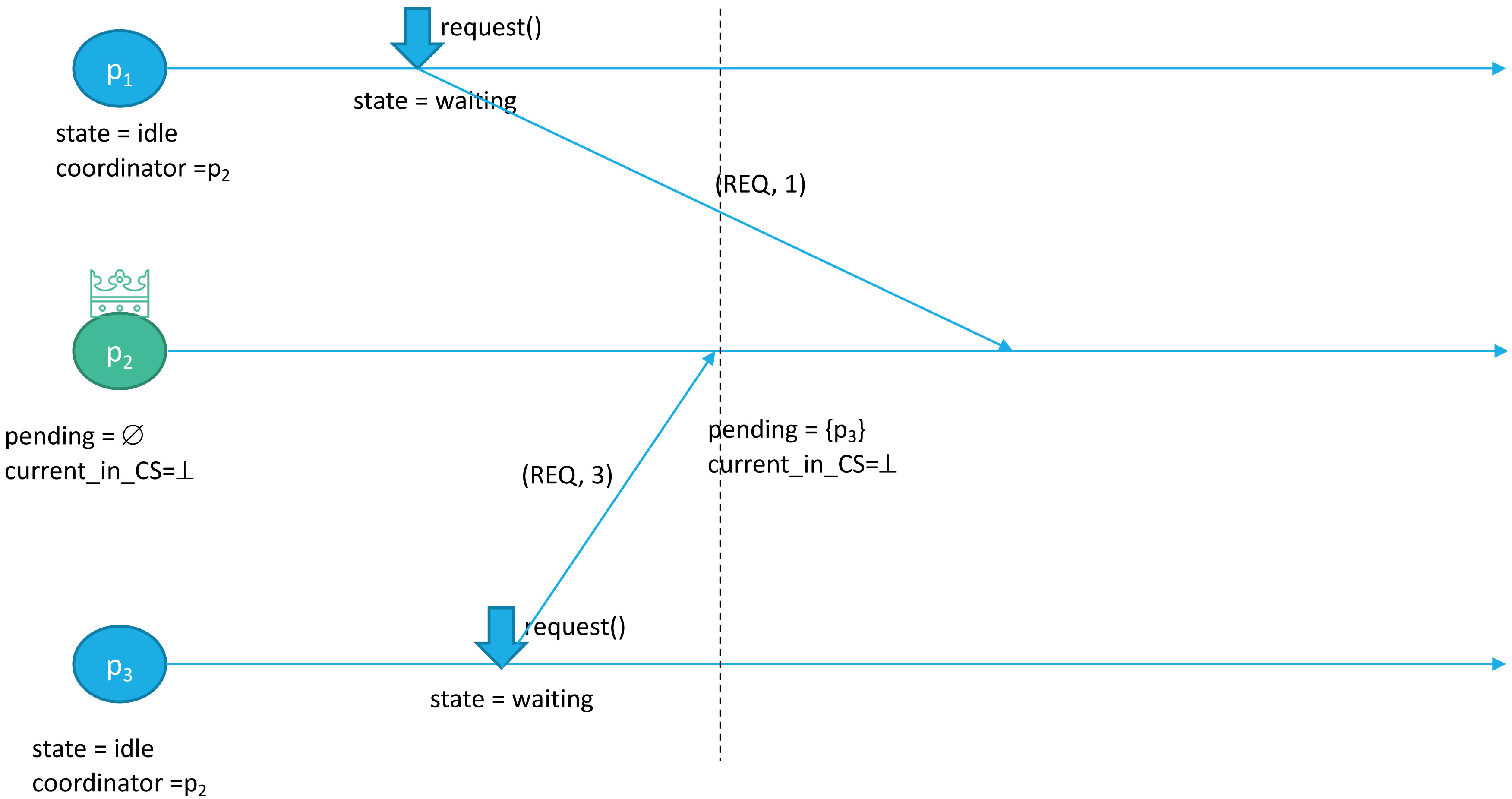


request()
p1
state = waiting
state = idle
coordinator =p2
(REQ, 1)
p2
pending = ∅
current_in_CS=⊥
p3
state = idle
coordinator =p2

Example

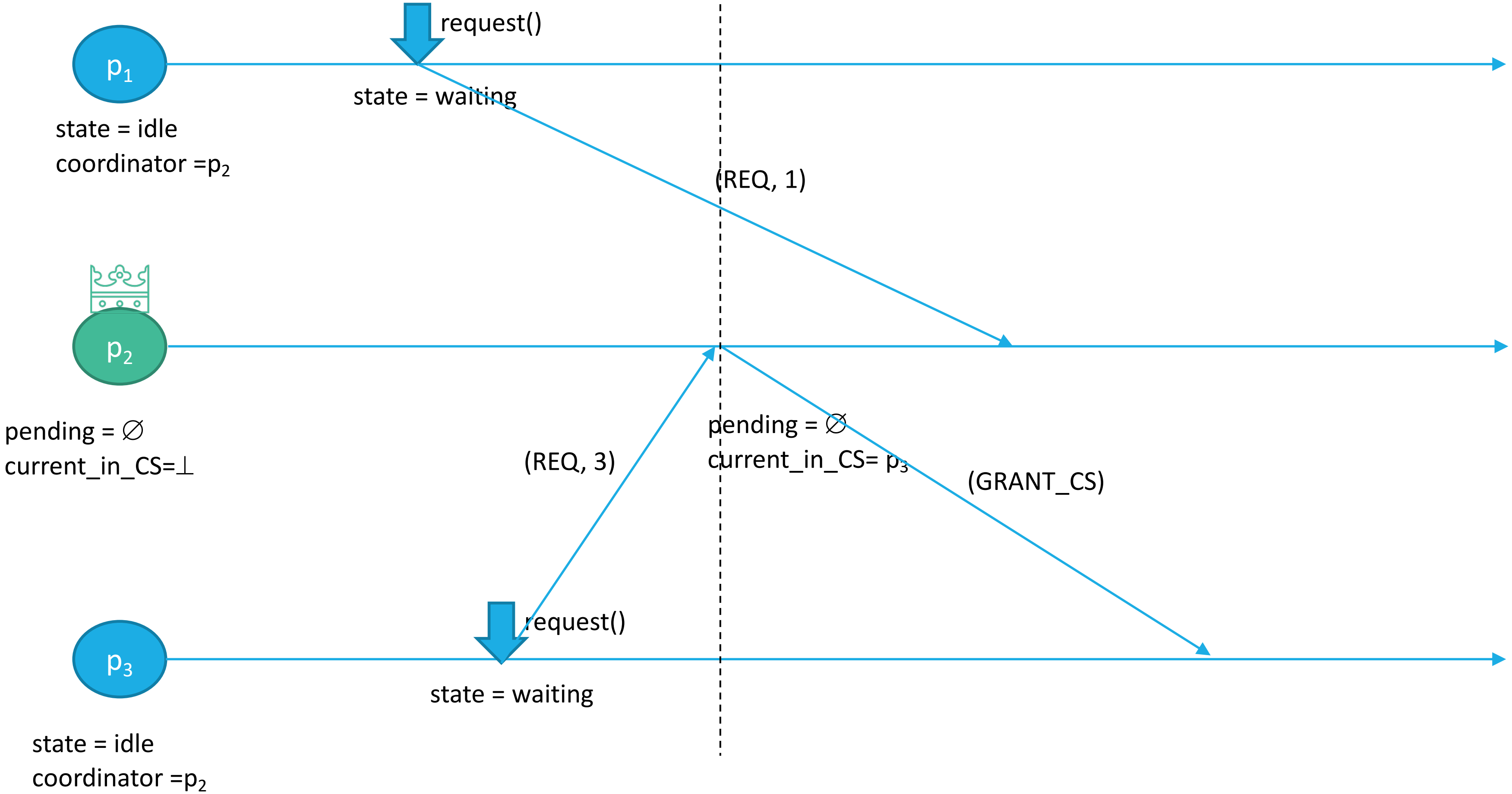

request()
p_1
state = waiting
state = idle
coordinator =p_2
(REQ, 1)
p_2
pending = $\varnothing$
current_in_CS=$\perp$
(REQ, 3)
request()
p_3
state = waiting
state = idle
coordinator =p_2

Example



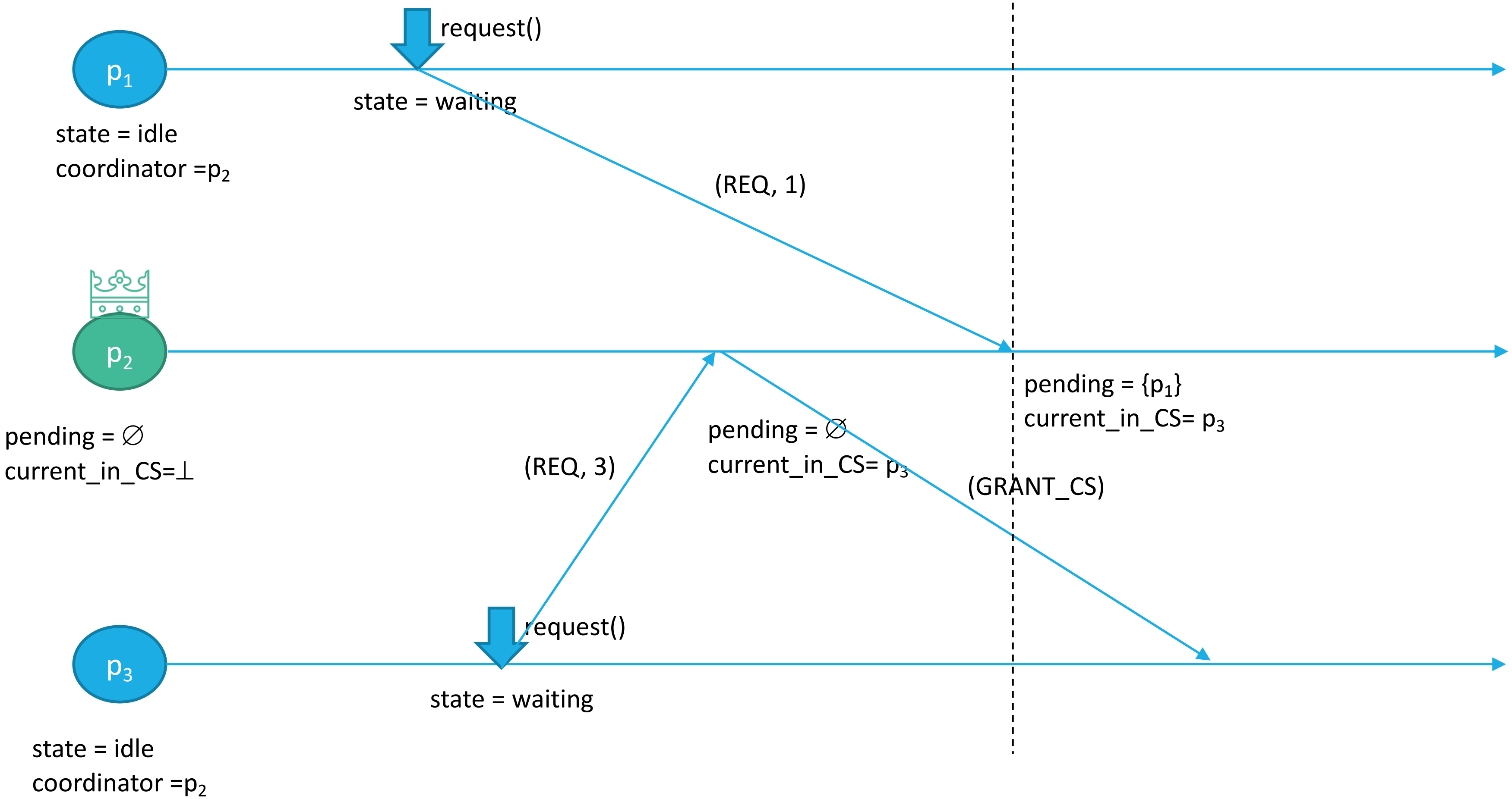
request()
p_1
state = waiting
state = idle
coordinator =p_2
(REQ, 1)
p_2
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = {p_3}
current_in_CS=⊥
request()
p_3
state = waiting
state = idle
coordinator =p_2

Example



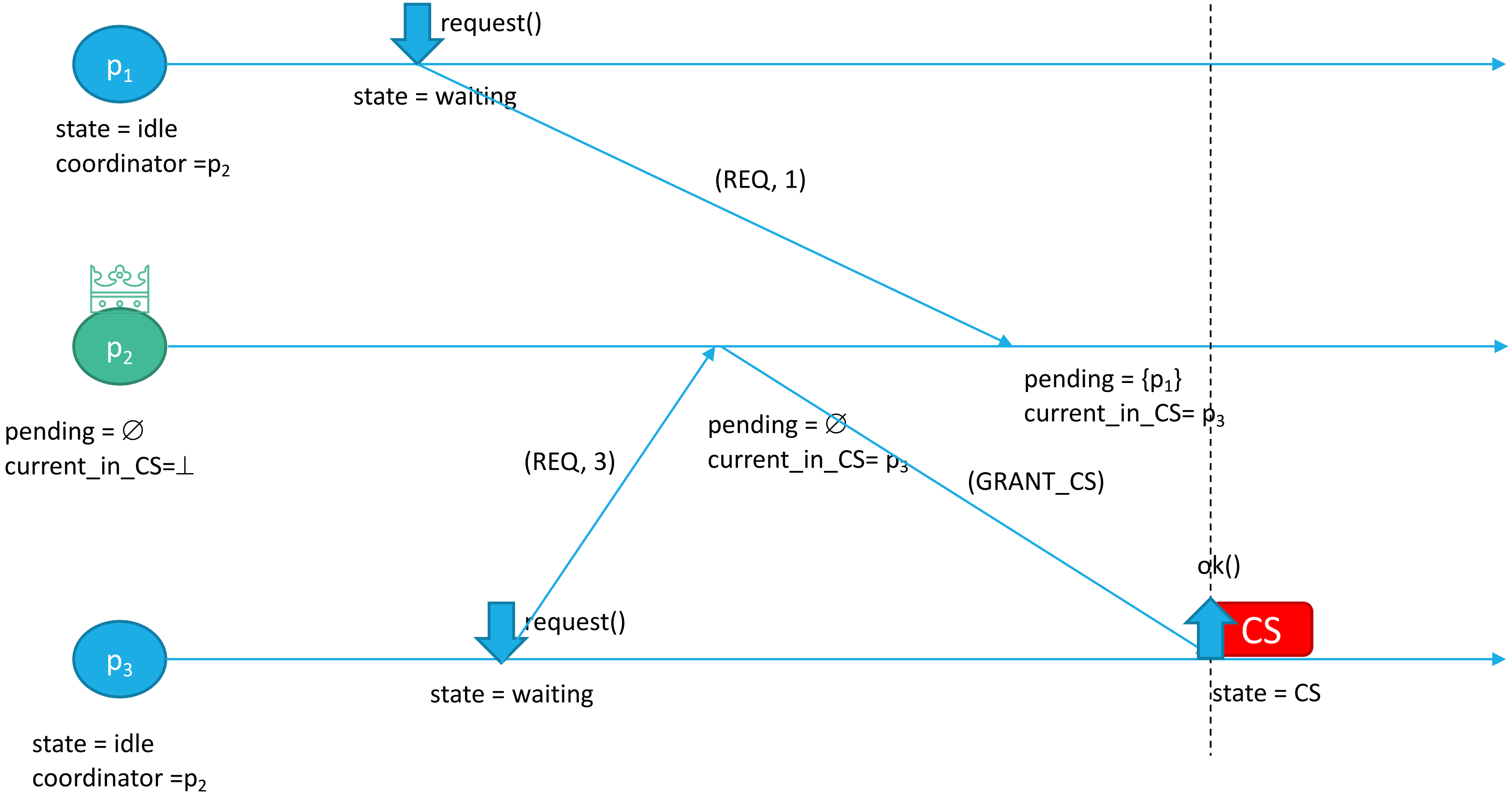
request()
p_1
state = waiting
state = idle
coordinator =p_2
(REQ, 1)
p_2
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = ∅
current_in_CS= p_3
(GRANT_CS)
request()
p_3
state = waiting
state = idle
coordinator =p_2

Example



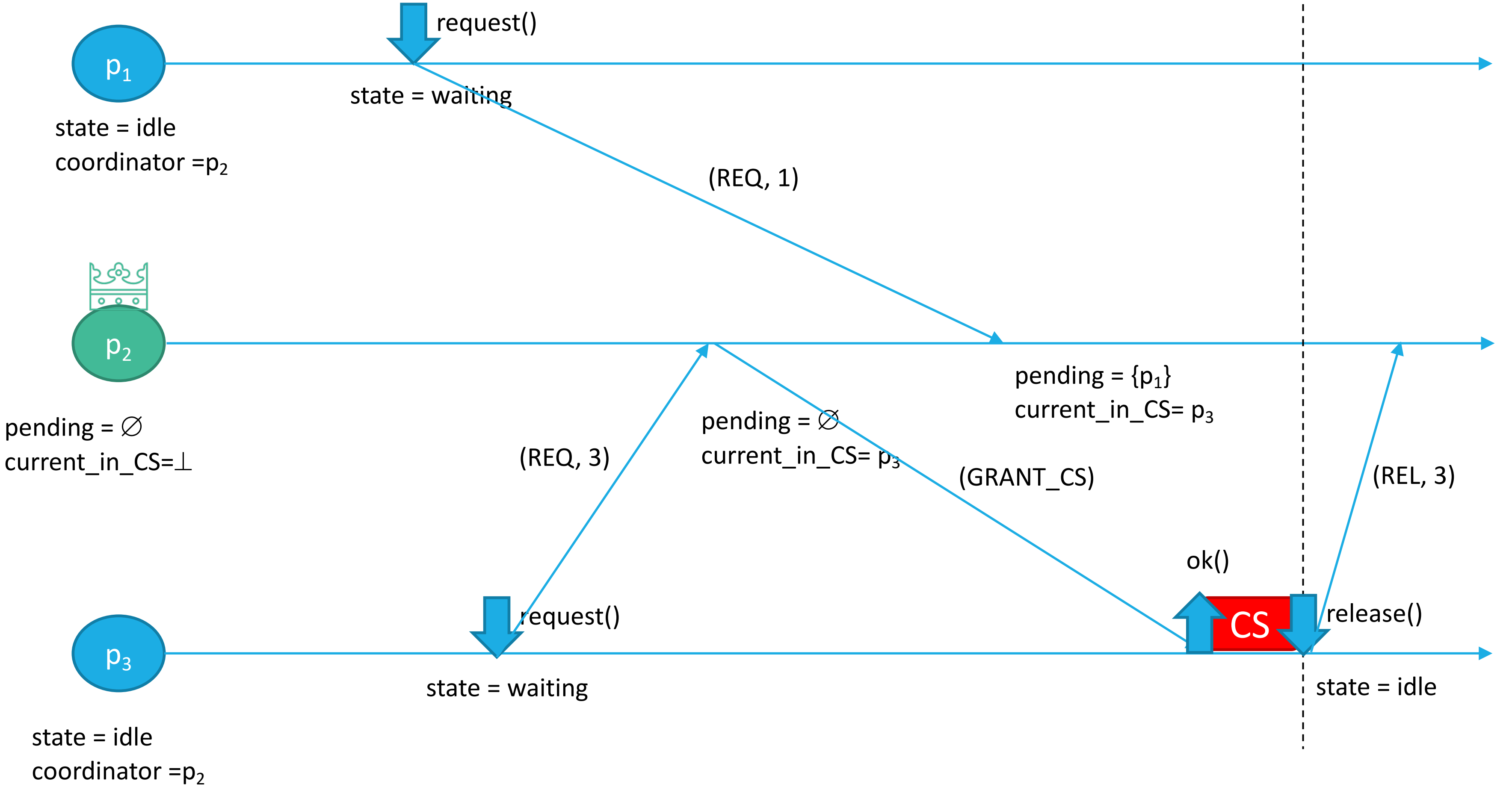
request()
p_1
state = waiting
state = idle
coordinator =p_2
(REQ, 1)
p_2
pending = {p_1}
current_in_CS= p_3
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = ∅
current_in_CS= p_3
(GRANT_CS)
request()
p_3
state = waiting
state = idle
coordinator =p_2

Example



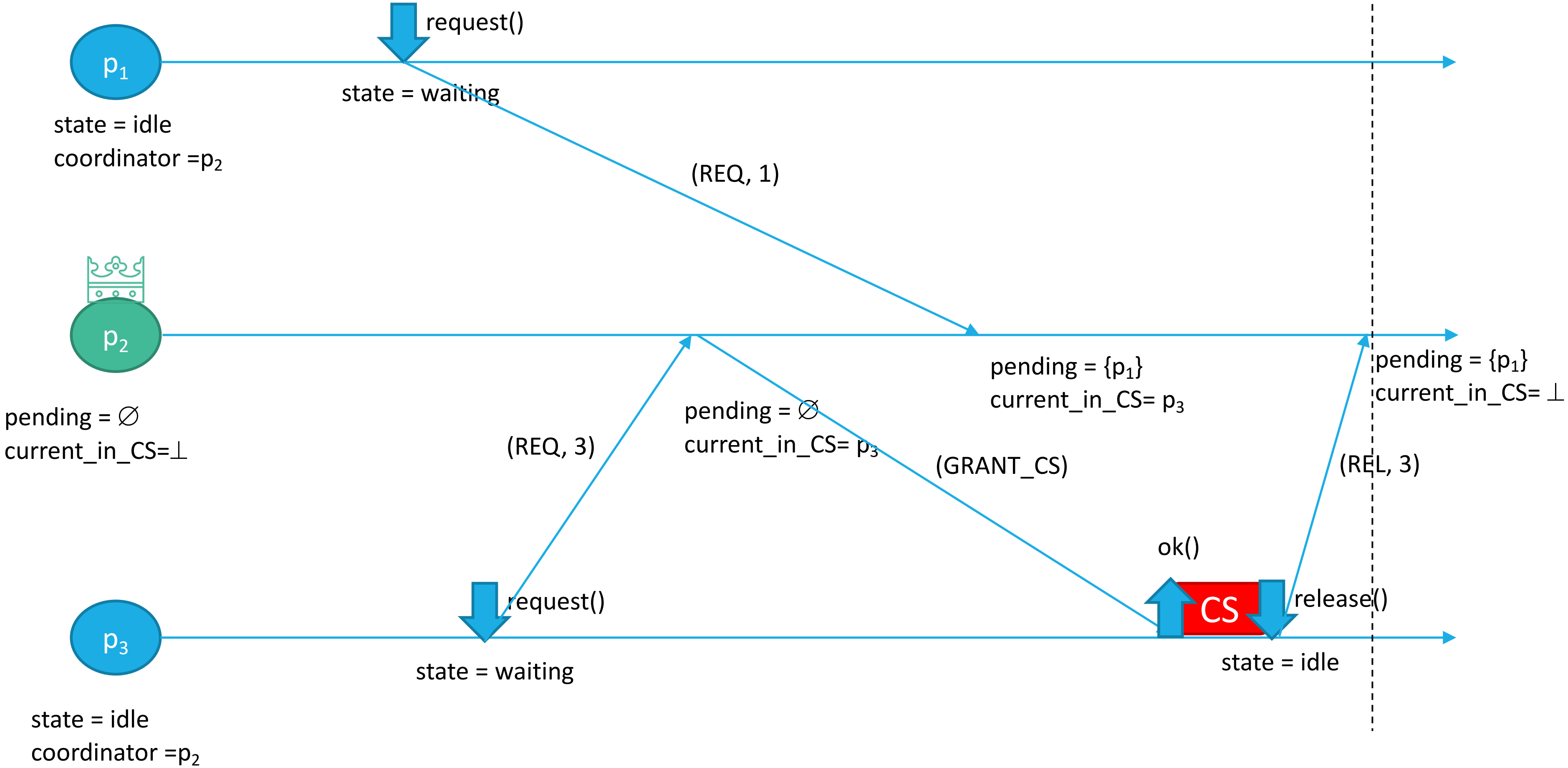
request()
p_1
state = waiting
state = idle
coordinator =p_2
(REQ, 1)
p_2
pending = {p_1}
current_in_CS= p_3
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = ∅
current_in_CS= p_3
(GRANT_CS)
ok()
request()
CS
p_3
state = waiting
state = CS
state = idle
coordinator =p_2

Example



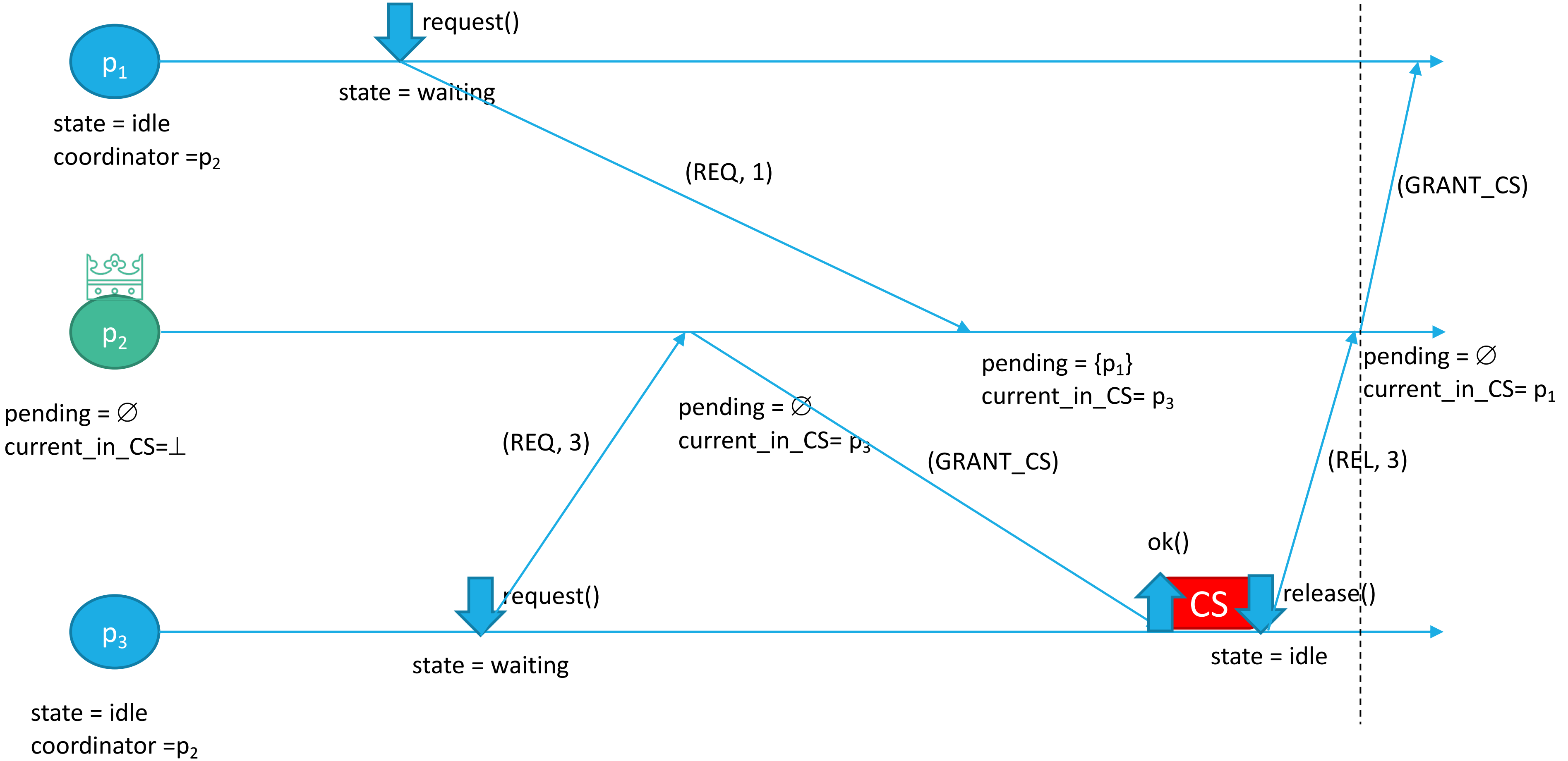
request()
p1
state = waiting
state = idle
coordinator =p2
(REQ, 1)
p2
pending = {p1}
current_in_CS= p3
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = ∅
current_in_CS= p3
(GRANT_CS)
(REL, 3)
ok()
CS
release()
request()
p3
state = waiting
state = idle
state = idle
coordinator =p2

Example



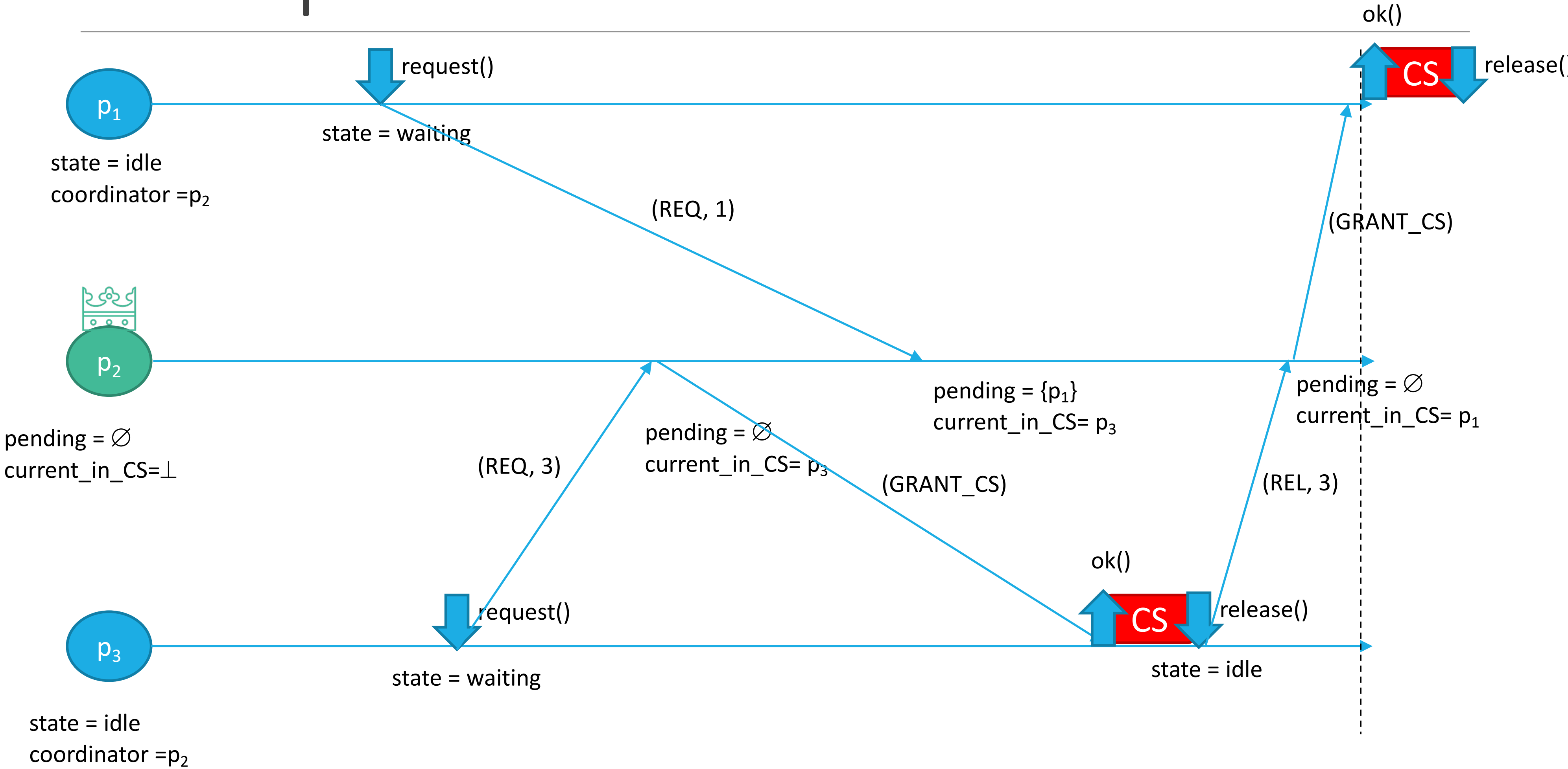
request()
p_1
state = waiting
state = idle
coordinator =p_2
(REQ, 1)
p_2
pending = {p_1}
current_in_CS= p_3
pending = {p_1}
current_in_CS= ⊥
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = ∅
current_in_CS= p_3
(GRANT_CS)
(REL, 3)
ok()
request()
CS
release()
p_3
state = idle
state = waiting
state = idle
coordinator =p_2

Example



request()
p_1
state = waiting
state = idle
coordinator =p_2
(REQ, 1)
(GRANT_CS)
p_2
pending = {p_1}
current_in_CS= p_3
pending = ∅
current_in_CS= p_1
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = ∅
current_in_CS= p_3
(GRANT_CS)
(REL, 3)
ok()
CS
release()
request()
p_3
state = waiting
state = idle
state = idle
coordinator =p_2

Example

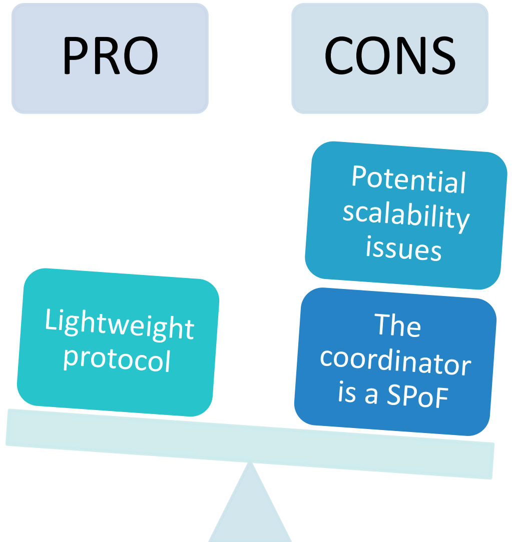


ok()
request()
CS
release()
p1
state = waiting
state = idle
coordinator =p2
(REQ, 1)
(GRANT_CS)
p2
pending = {p1}
current_in_CS= p3
pending = ∅
current_in_CS= p1
pending = ∅
current_in_CS=⊥
(REQ, 3)
pending = ∅
current_in_CS= p3
(GRANT_CS)
(REL, 3)
ok()
CS
request()
release()
p3
state = waiting
state = idle
state = idle
coordinator =p2

Discussion

PERFORMANCE

- entering the CS always requires 2 messages (i.e., REQ and GRANT) taking one RTT
- releasing the CS only requires 1 message
  - such message represent the delay between two different accesses to the CS



Token-based Algorithm on Logical Ring

BASIC IDEA

- a process interested to the CS can access it only when it receives a token
- The token is unique and it is exchanged between processes
- to guarantee fairness, we can exploit a structured logical topology (i.e., a ring) for exchanging messages related to the mutual exclusion protocol

Token-based Algorithm on Logical Ring

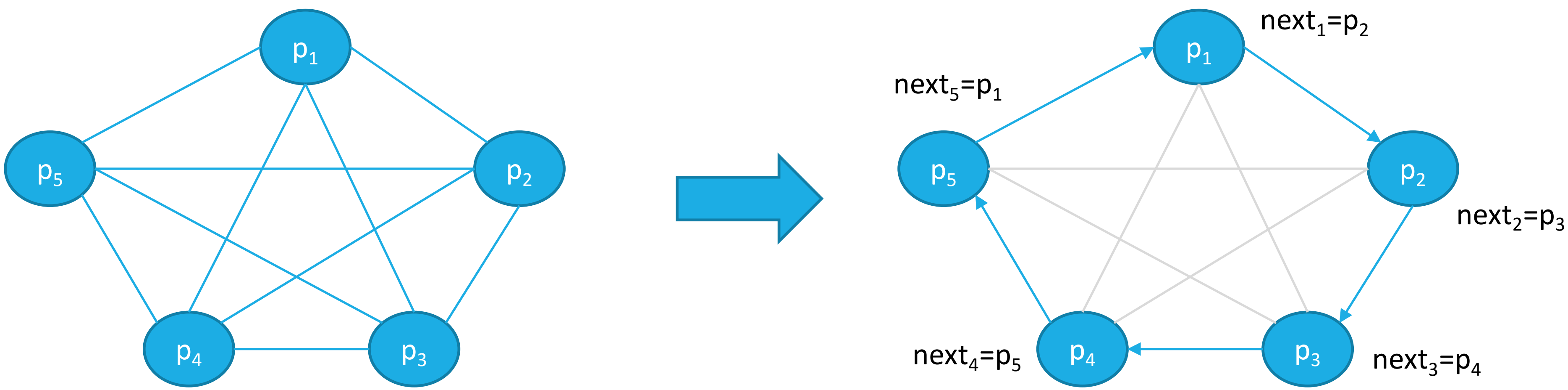
INTUITION OF THE ALGORITHM

1. we construct an overlay (i.e., a logical network) as a ring exploiting existing point-to-point communication channels
2. A token is created and inserted in the ring during the initialization phase (i.e., it is assigned to a process of the system)
3. When a process requests the CS
   a. it waits until it gets the token
   b. enter the CS and upon release it sends the token to its next on the ring
4. If a process receives the token and it is not interested in the CS, it simply passes it to the next in the ring

Token-based Algorithm on Logical Ring

INTUITION

1. we construct an overlay (i.e., a logical network) as a ring exploiting existing point-to-point communication channels
   - The ring is obtained by:
     - storing in a local variable the name of the next process in the ring and
     - allowing the communication only with the next

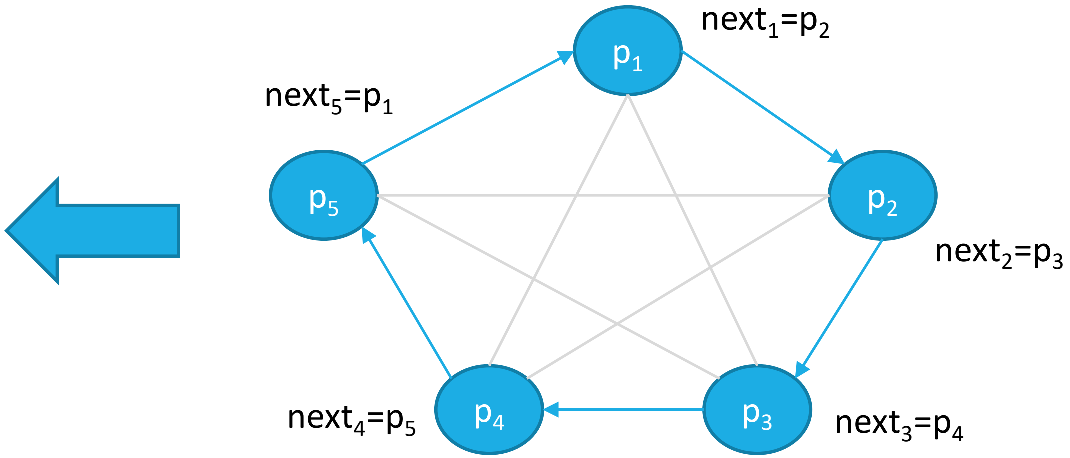


Token-based Algorithm on Logical Ring

INTUITION

1. we construct an overlay (i.e., a logical network) as a ring exploiting existing point-to-point communication channels
   - The ring is obtained by:
     - storing in a local variable the name of the next process in the ring and
     - allowing the communication only with the next

Init
state = idle
next = $p_{(i+1) \bmod N}$
…



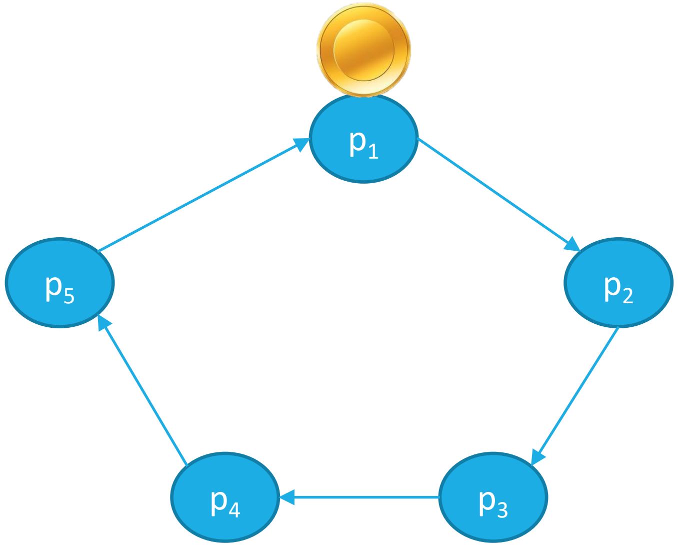
Token-based Algorithm on Logical Ring

INTUITION OF THE ALGORITHM

2. A token is created and propagated in the ring during the initialization phase (i.e., it is assigned to a process of the system)

WARNING: The token must be unique to guarantee mutual exclusion

Only one process (selected trough a deterministic function) can create the token during the init



Init

state = idle

next = $p_{(i+1) \bmod N}$

if self = p_0

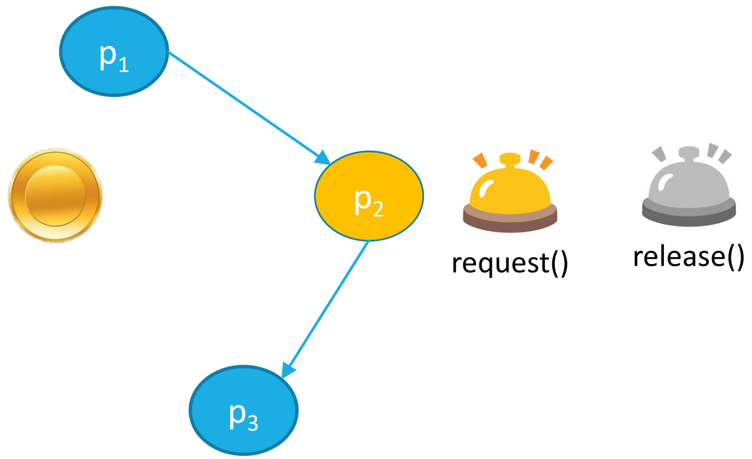
    trigger pp2pSend(TOKEN) to next

…

Token-based Algorithm on Logical Ring

INTUITION OF THE ALGORITHM

3. When a process requests the CS
   a. it waits until it gets the token
   b. enter the CS and upon release it sends the token to its next on the ring
4. If a process receives the token and it is not interested in the CS, it simply passes it to the next in the ring




```
upon event request()
        state = waiting

upon event pp2pDeliver(TOKEN)
        if state == waiting
                state = CS
                trigger ok()
        else
                trigger pp2pSend(TOKEN) to next
upon event release()
        state = idle
        trigger pp2pSend(TOKEN) to next
```

Token-based Algorithm on Logical Ring

```
Init
state = idle
next = p_(i+1)mod N
if self = p_0
        trigger pp2pSend(TOKEN) to next


upon event request()
        state = waiting

upon event pp2pDeliver(TOKEN)
        if state == waiting
                state = CS
                trigger ok()
        else
                trigger pp2pSend(TOKEN) to next

upon event release()
        state = idle
        trigger pp2pSend(TOKEN) to next
```

Discussion

The algorithm continuously consume communication resources (even if no one is interested to the CS)

The delay experienced by every process between the request and the grant varies between 0 (it just receives the token) and N messages (it just forwarded the token)

Quorum-based Algorithm – Maekawa's voting algorithm

BASIC IDEA

- to enter the CS every process waits to get the acknowledgement only by a subset of processes large enough to guarantee conflicts

Each process p_i has associated a *voting set* V_i

- Voting sets must satisfy the following properties
  - $p_i \in V_i$
  - $\forall$ i, j, $V_i \cap V_j \neq \emptyset$ (i.e., there is at least one common member for each pair of voting sets)
  - $|V_i| = K$ (voting sets have all the same size for fairness – same load principle)
  - each p_j is contained in M voting sets (same responsibility principle)

Quorum-based Algorithm – Maekawa’s voting algorithm

```
Init
state = released
voted = false
V_i = get_voting_set(i)
replies = ∅
pending = ∅

upon event request()
        state = wanted
        for each p_j ∈(V_i \p_i) do
                trigger pp2pSend(REQ, i) to p_j


upon event pp2pDeliver(REQ, j)
        if state == held OR voted == true
                pending = pending ∪ {i}
        else
                trigger pp2pSend(ACK, i) to p_j
                voted = true
```

```
upon event pp2pDeliver(ACK, j)
        replies = replies ∪ {j}

when |replies| == |V_i|-1
        state = held

upon event release()
        state = released
        replies = ∅
        for each p_j ∈(V_i \p_i) do
                trigger pp2pSend(REL, i) to p_j


upon event pp2pDeliver(REL, j)
        if |pending| > 0
                candidate=select_next(pending)
                pending = pending \candidate
                trigger pp2pSend(ACK, i) to candidate
                voted = true
        else
                voted = false
```

Quorum-based Algorithm – Maekawa's voting algorithm

CHALLENGE: How to compute the values K and M to balance load and responsibilities?

- Maekawa showed that the optimal solution which minimize k and allows to get ME is having
  - K ~ $\sqrt[2]{N}$
  - M = K

- An approximation to define V_i is having sets such that $|V_i|$ ~ $2\sqrt[2]{N}$ defined as follows
  - Place processes in a matrix of size $\sqrt[2]{N}$ x $\sqrt[2]{N}$
  - for each p_i, let V_i be the union of the rows and columns containing p_i

Example

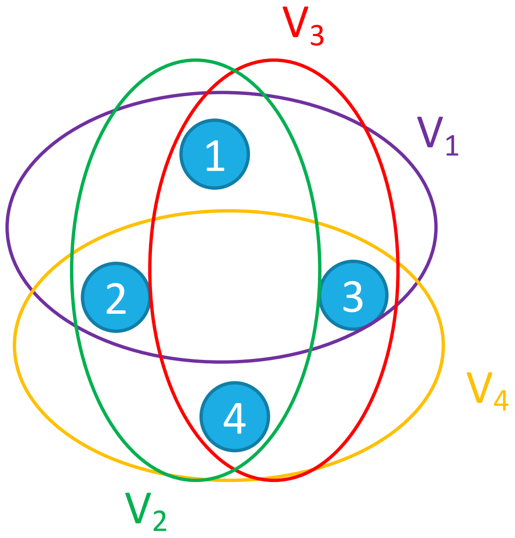
Let us consider a system composed by N = 4 processes

- $(M = K \sim \sqrt[2]{4}$ and $|V_i| \sim 2\sqrt[2]{4})$

Let's place processes in the matrix and compute V_i

p1	p2
p3	p4

Process id	**V**
1	p_1, p_2, p_3
2	p_1, p_2, p_4
3	p_1, p_3, p_4
4	p_2, p_3, p_4



Example

Let us consider a system composed by N = 9 processes

- (M = K ~ $\sqrt[2]{9}$ and |Vi|~ $2\sqrt[2]{9} = 6$)

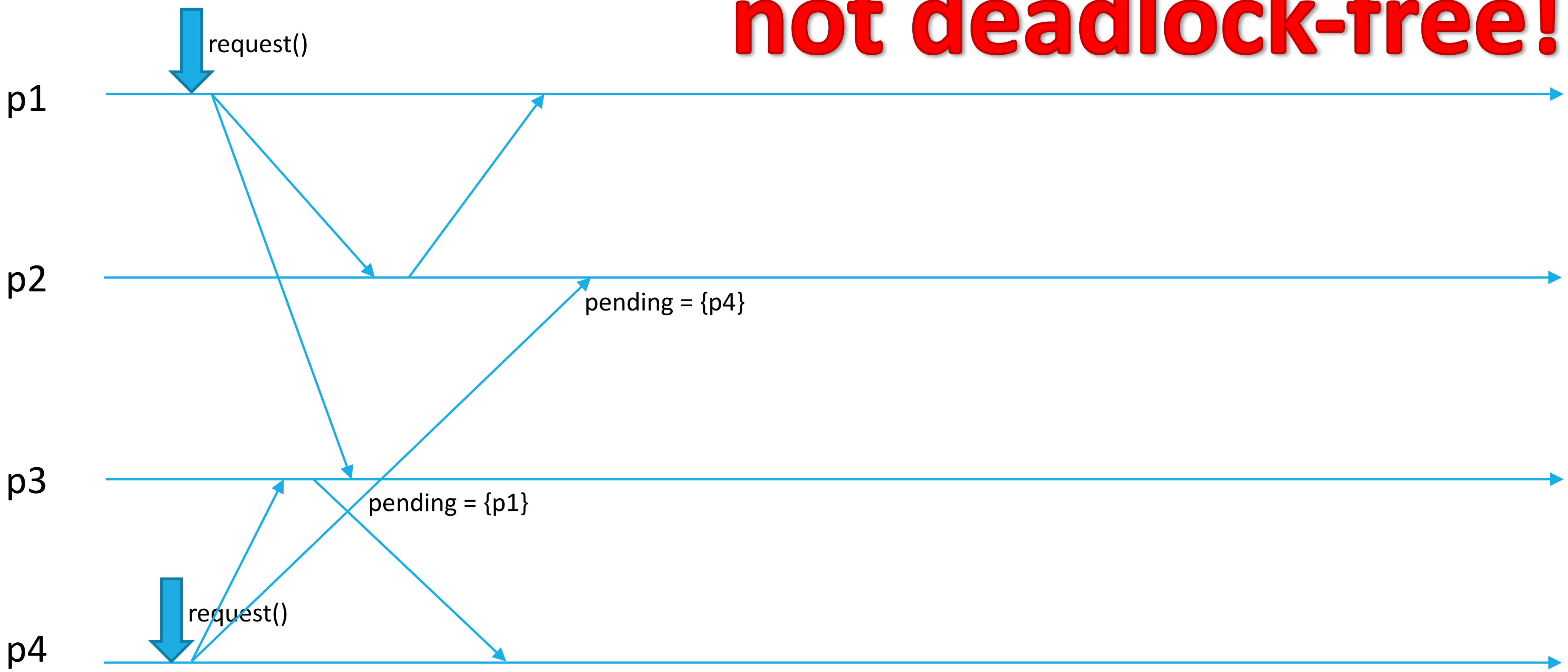
Let's place processes in the matrix

p1	p2	p3
p4	p5	p6
p7	p8	p9

Process id	V
1	p_1, p_2, p_3, p_4, p_7
2	p_1, p_2, p_3, p_5, p_8
3	p_1, p_2, p_3, p_6, p_9
4	p_4, p_5, p_6, p_1, p_7
5	p_4, p_5, p_6, p_2, p_8
6	p_4, p_5, p_6, p_3, p_9
7	p_7, p_8, p_9, p_1, p_4
8	p_7, p_8, p_9, p_2, p_5
9	p_7, p_8, p_9, p_3, p_6

Example

WARNING!
Maekawa's Algorithm is not deadlock-free!



Reference

George Coulouris, Jean Dollimore and Tim Kindberg, Gordon Blair "Distributed Systems: Concepts and Design (5th Edition)". Addison - Wesley, 2012.

- Chapter 11 – section 11.2