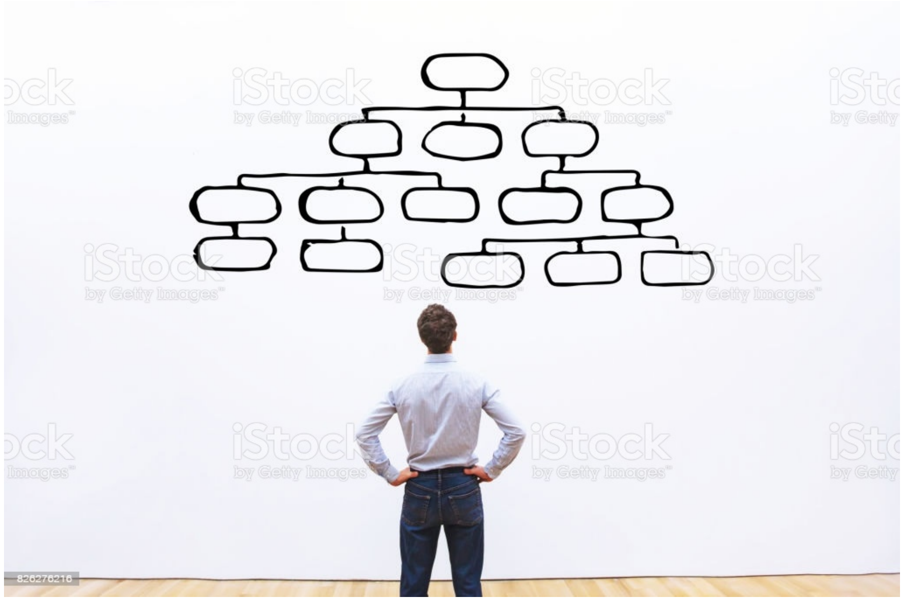


Pixabay license

DISTRIBUTED SYSTEM MODELS

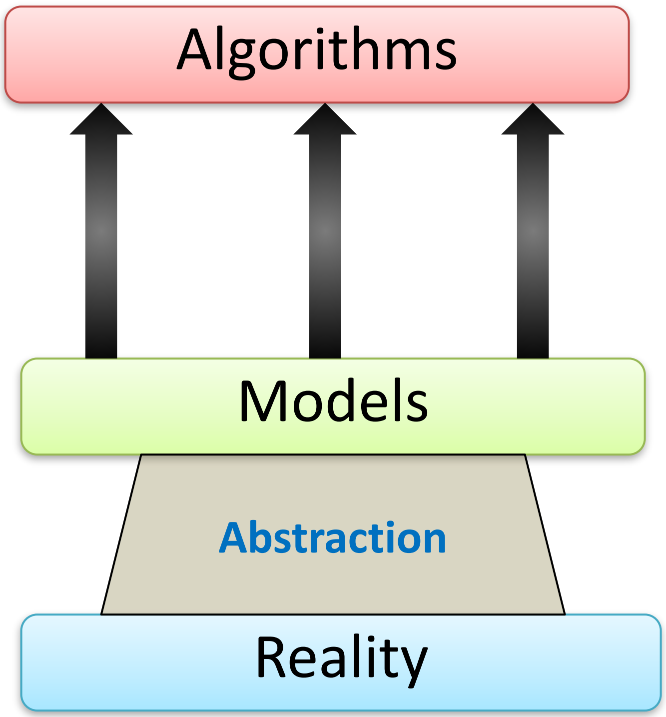
Distributed system model

Of theoretical relevance for designing algorithms

- Model captures all the **assumptions** made about the system
- Including network, clock, processor, etc.
- Algorithms always **assume** a certain model
- Some algorithms are only possible in **stronger models** (more restrictions)
- Model is of **theoretical relevance** whether its assumptions hold in practice is a different question

"All models are wrong, but some are useful"

George Box, ~1976



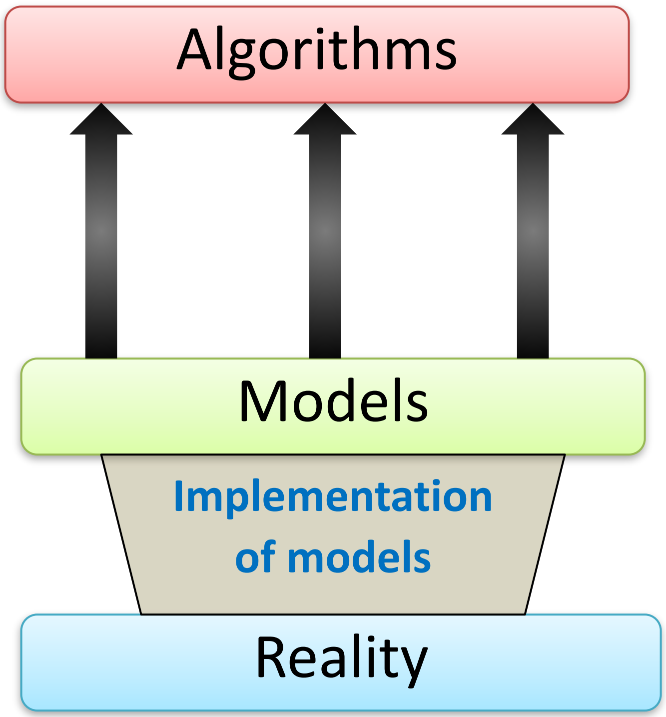
Distributed system model

Of theoretical relevance for designing algorithms

- Model captures all the **assumptions** made about the system
- Including network, clock, processor, etc.
- Algorithms always **assume** a certain model
- Some algorithms are only possible in **stronger models** (more restrictions)
- Model is of **theoretical relevance** whether its assumptions hold in practice is a different question

"All models are wrong, but some are useful"

George Box, ~1976



Synchronous vs. asynchronous model

Property	Synchronous system model	Asynchronous system model
Clocks	Bound on drift	No bound on drift
Processor	Bound on execution time	No bound on execution time
Channel	Bound on latency	No bound on latency

Synchronous distributed system model

- Each process has a local clock whose drift rate from real time is **within a known bound**
- Each step of a computation will complete **within a known bound**
- Each message transmitted over a channel is received **within a known bound**

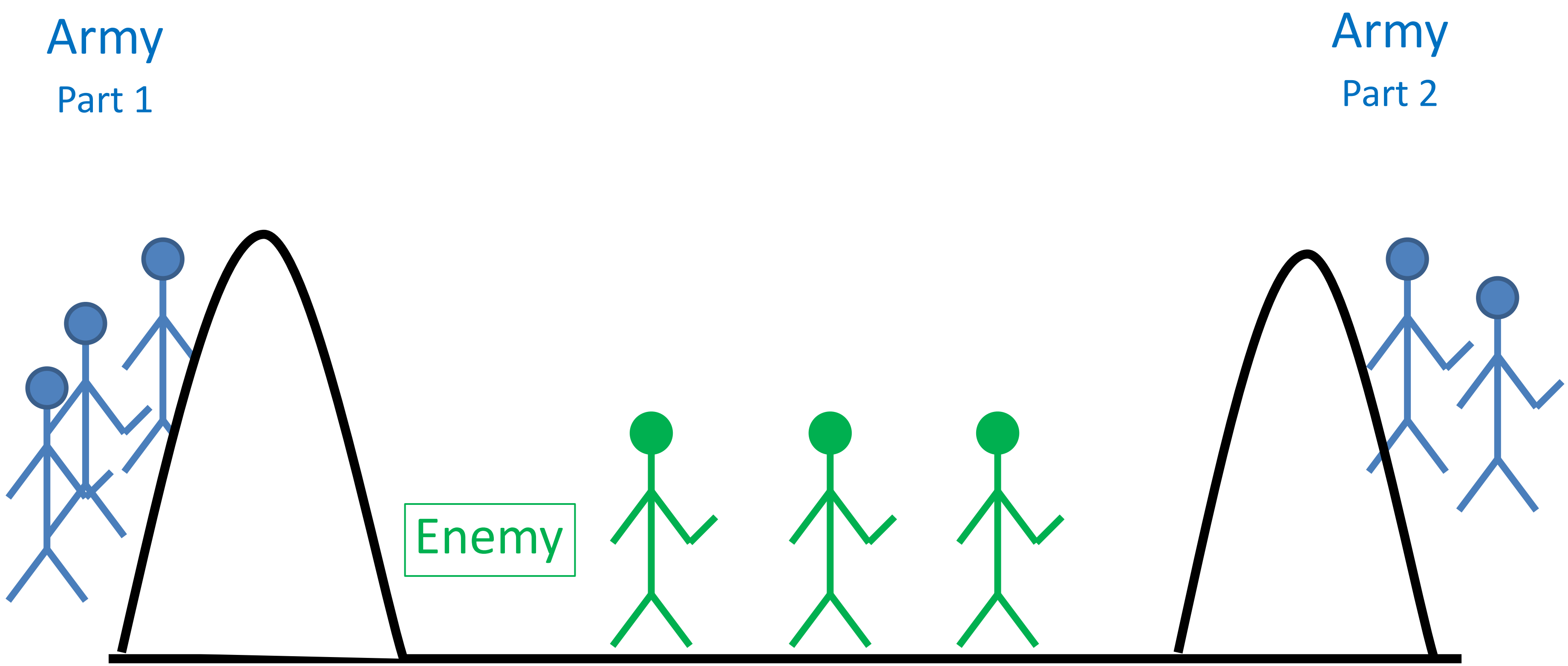
Asynchronous distributed system model

- Clock **drift rates are arbitrary**
- Each step of a computation may take an **arbitrary time to complete** (but will eventually complete)
- Messages may need an **arbitrary time to be transmitted** (but will eventually be transmitted)

Two General's Problem (Agreement)

Armies need to agree on:

A thought experiment



Armies are safe if they don't attack (or win) (Safety)

Armies need to coordinate attack to win. (Liveness)

Two General's Problem (Agreement)

Armies need to agree on:

A thought experiment

Who leads the attack?

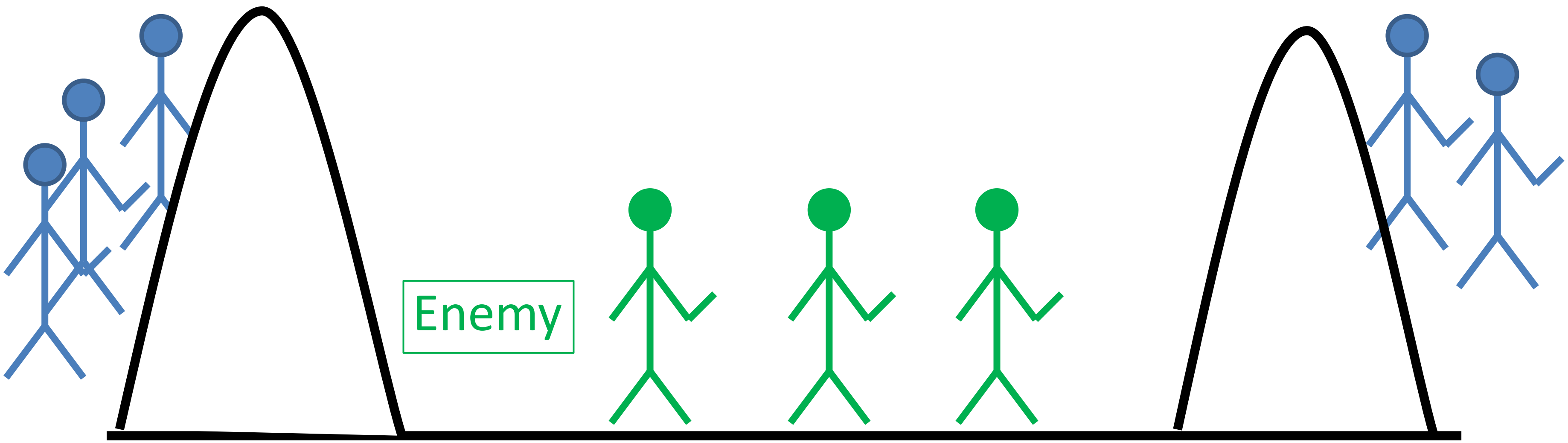
When to attack?

Army

Part 1

Army

Part 2



Armies are safe if they don't attack (or win) (Safety)

Armies need to coordinate attack to win. (Liveness)

Two General's Problem (Agreement)

Armies need to agree on:

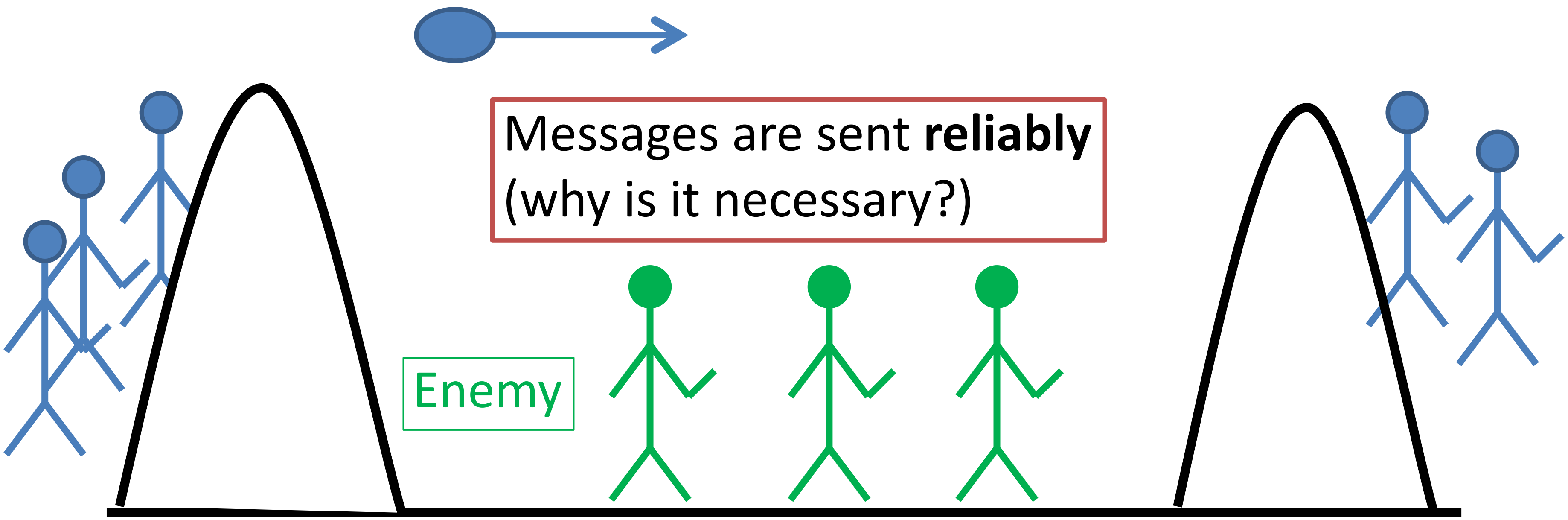
A thought experiment

Who leads the attack?

When to attack?

Army
Part 1

Army
Part 2



Messages are sent **reliably** (why is it necessary?)

Armies are safe if they don't attack (or win) (Safety)

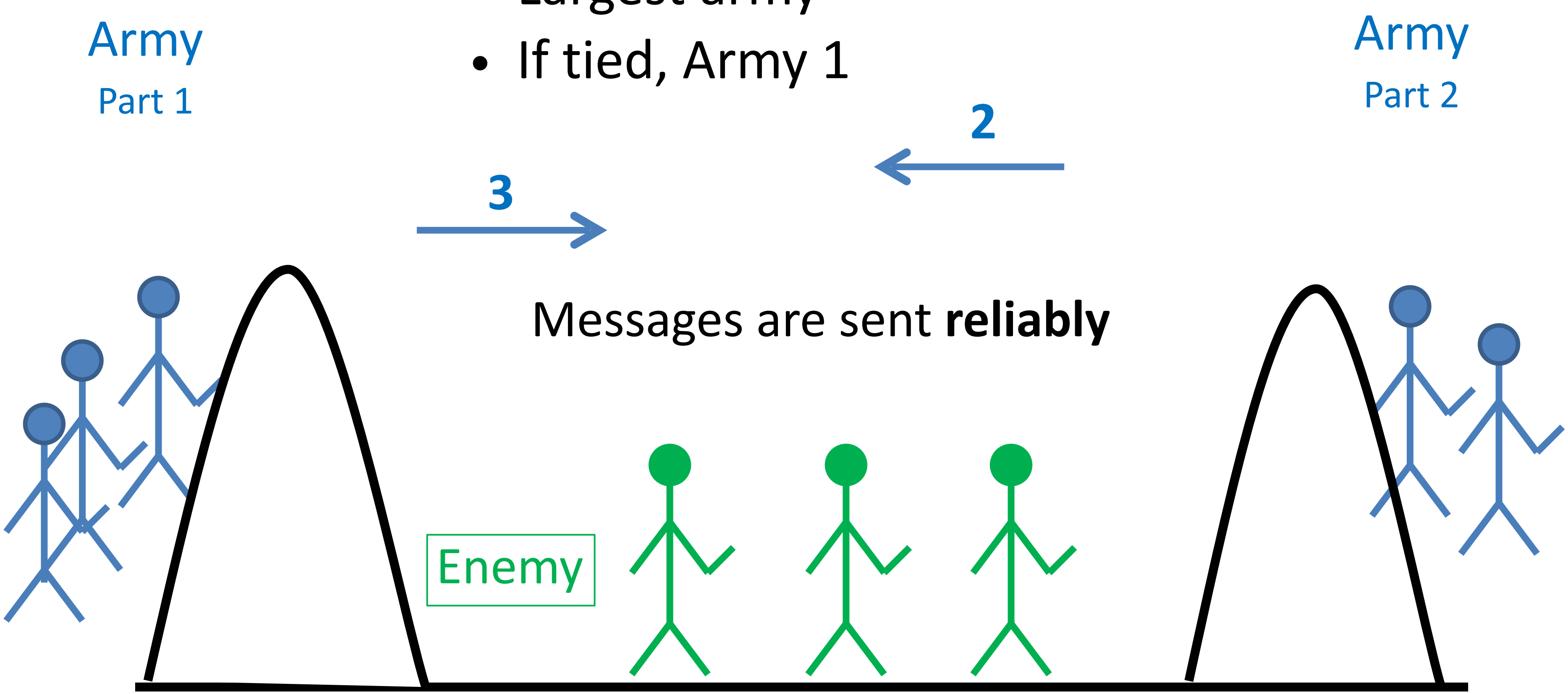
Armies need to coordinate attack to win. (Liveness)

Synchronous vs. asynchronous agreement

Who leads the attack?

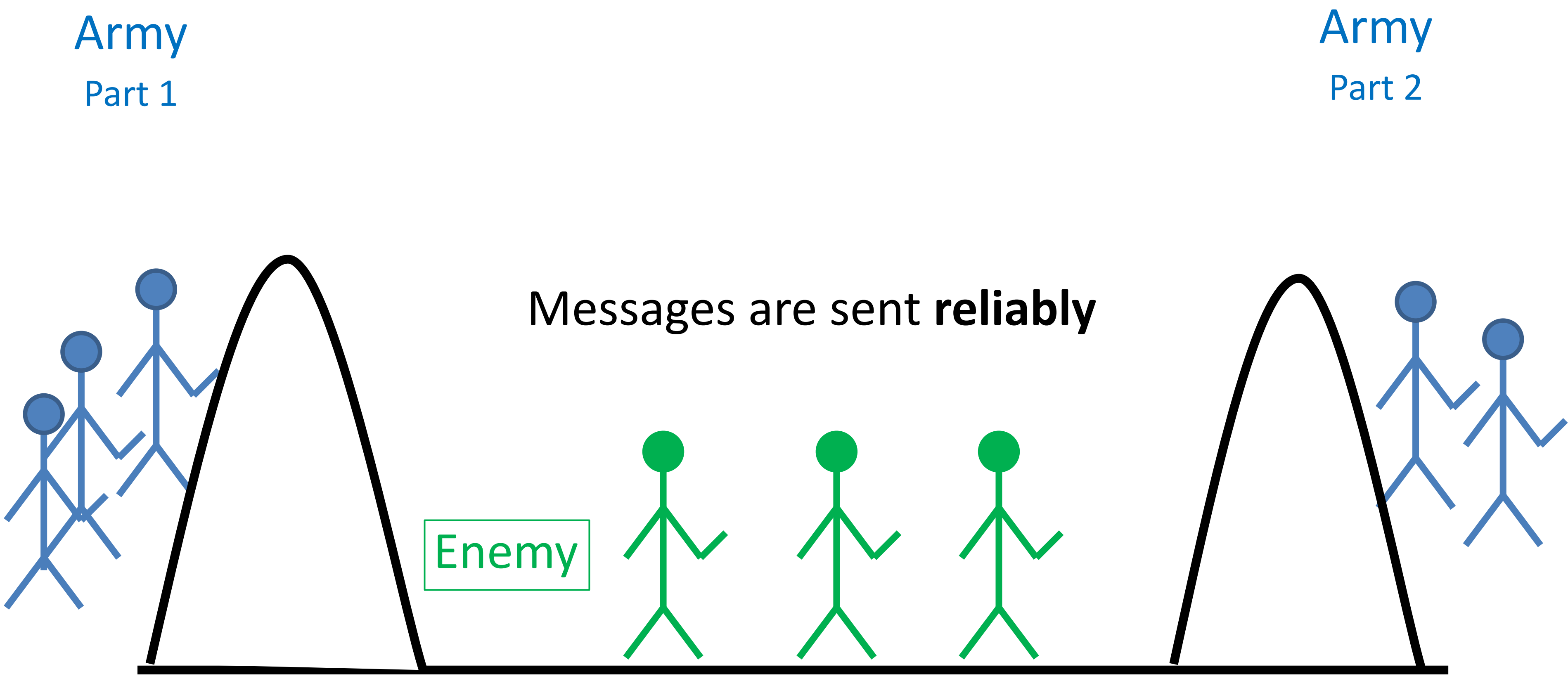
- Largest army
- If tied, Army 1

Messages are sent **reliably**



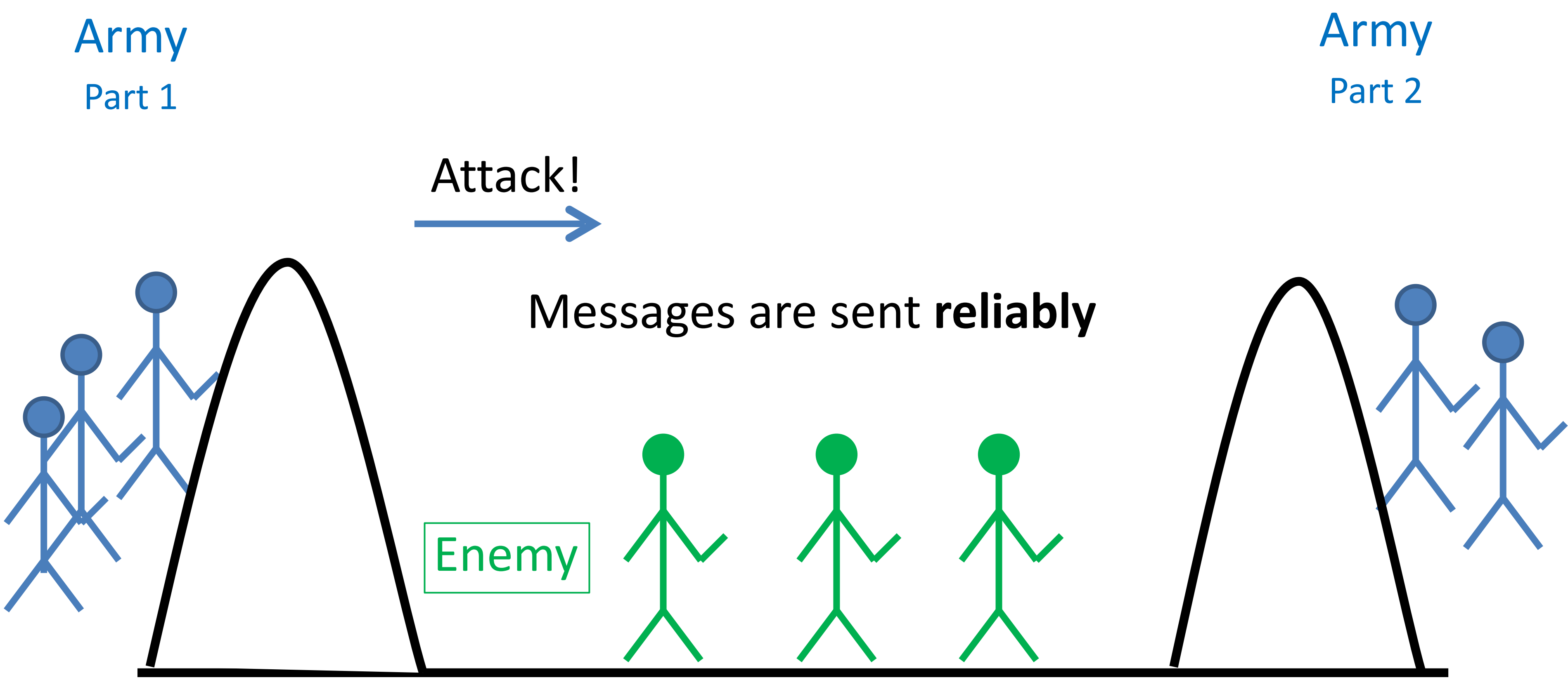
Asynchronous agreement

When to attack? No bound on delivery!



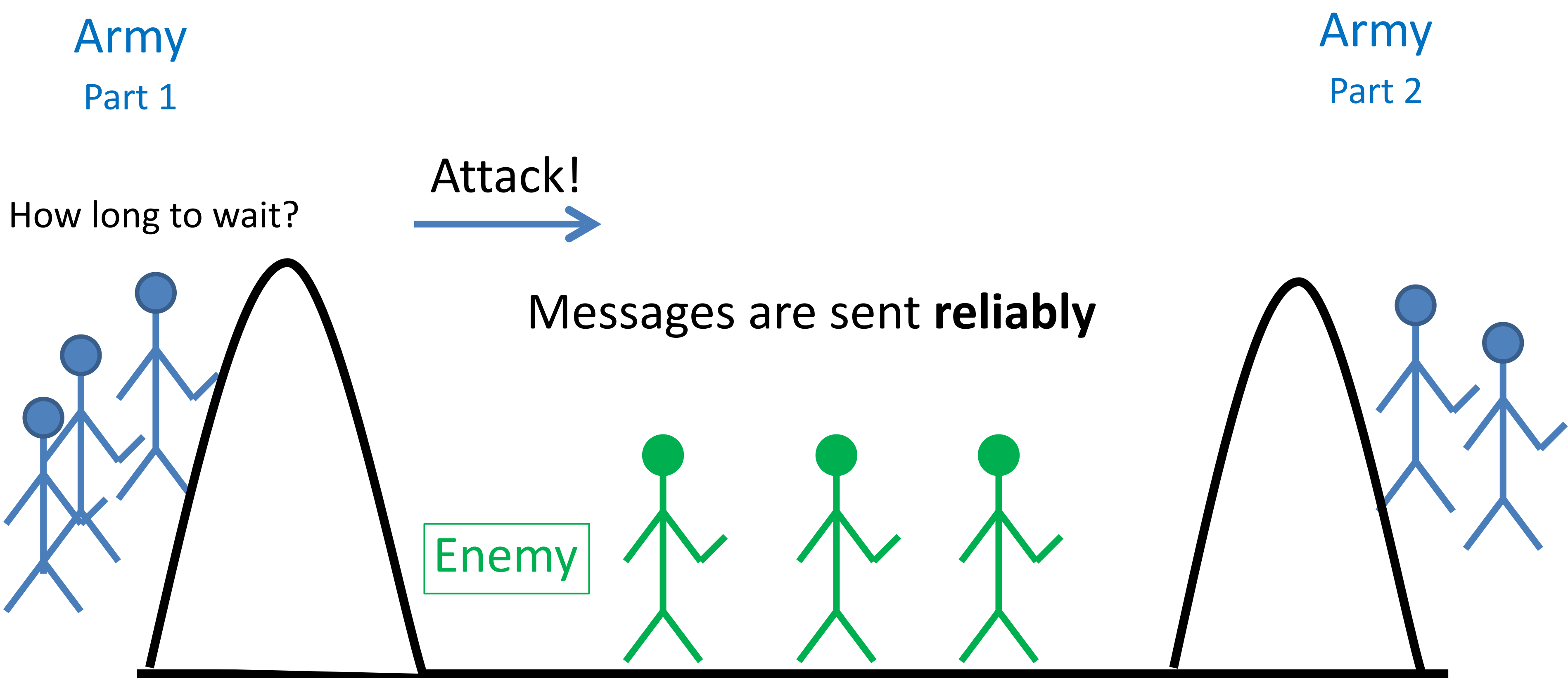
Asynchronous agreement

When to attack? No bound on delivery!



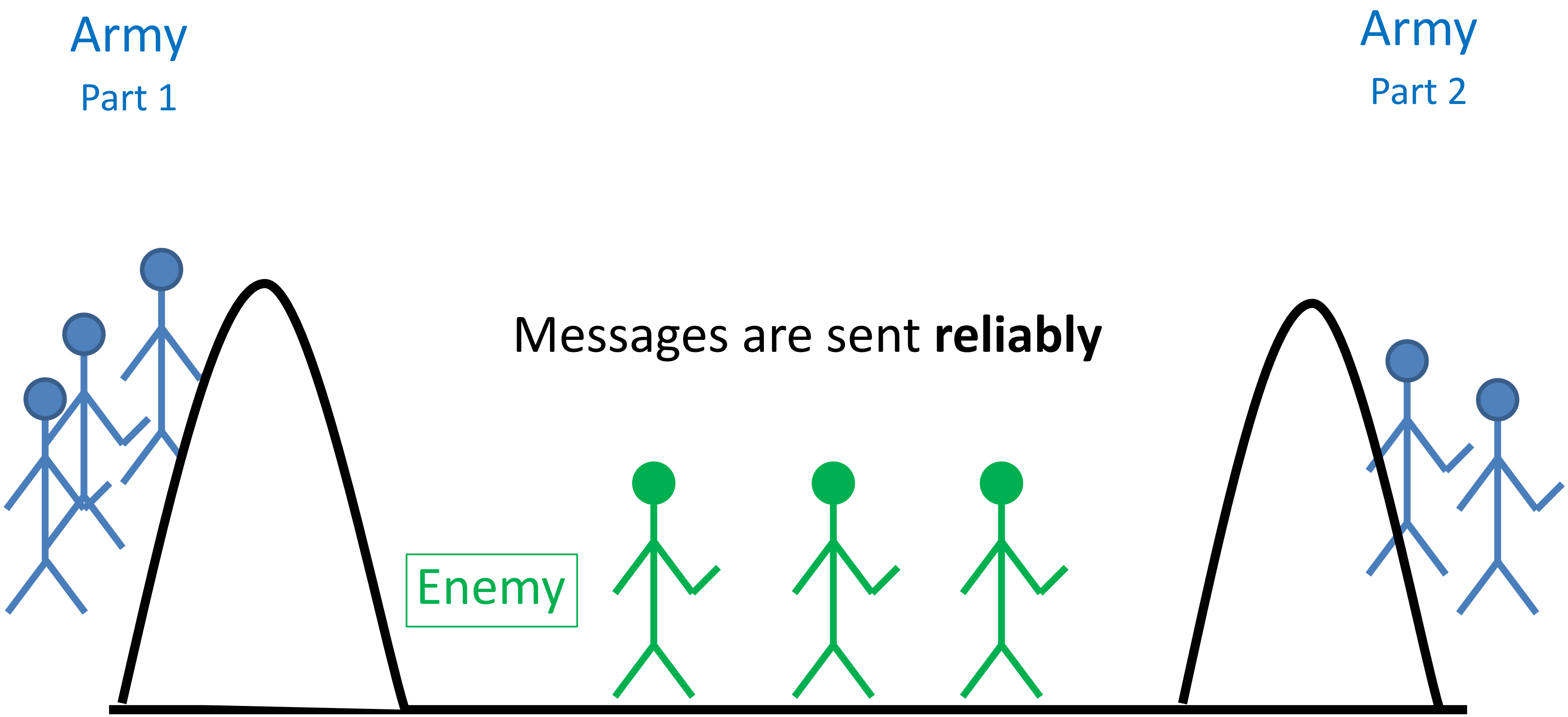
Asynchronous agreement

When to attack? No bound on delivery!



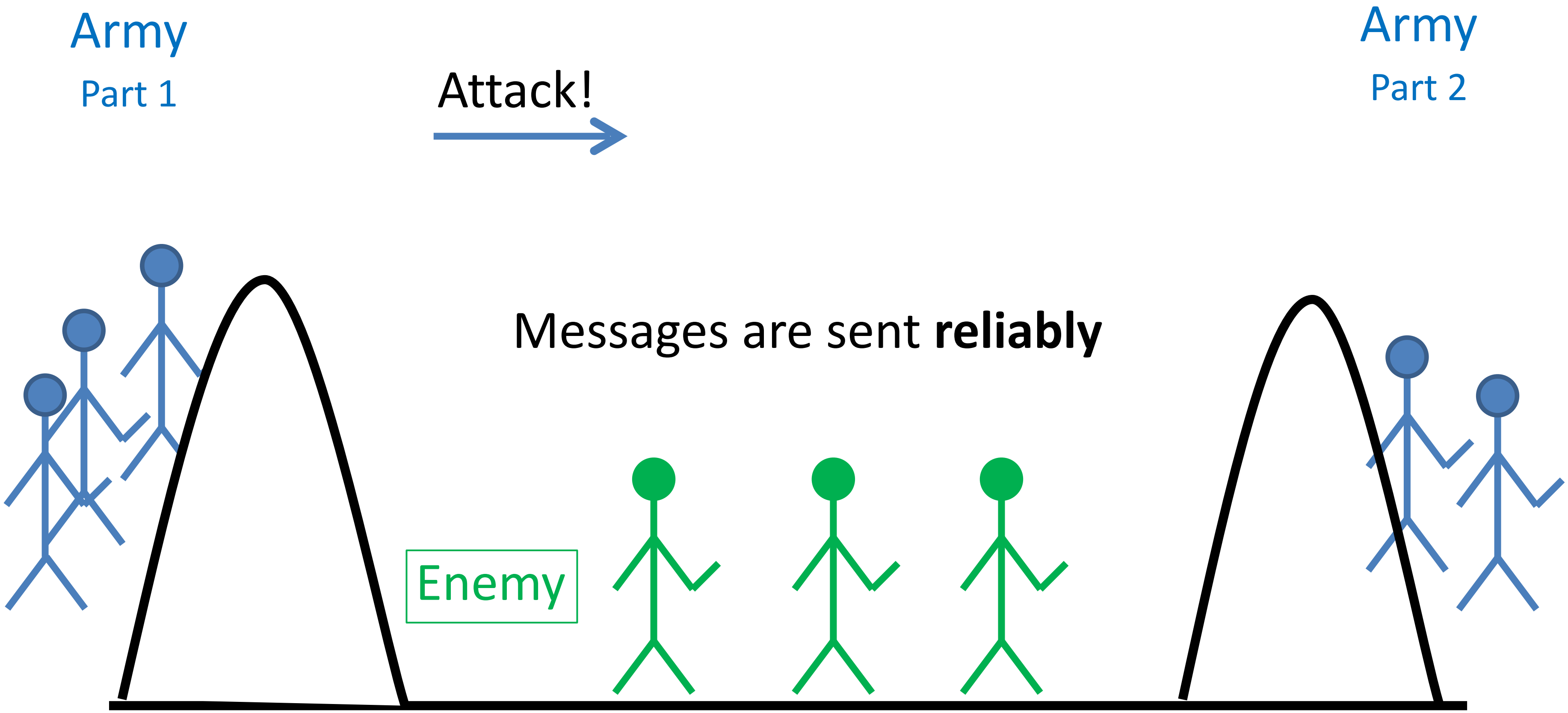
Synchronous agreement

When to attack?



Synchronous agreement

When to attack?



Synchronous agreement

When to attack?

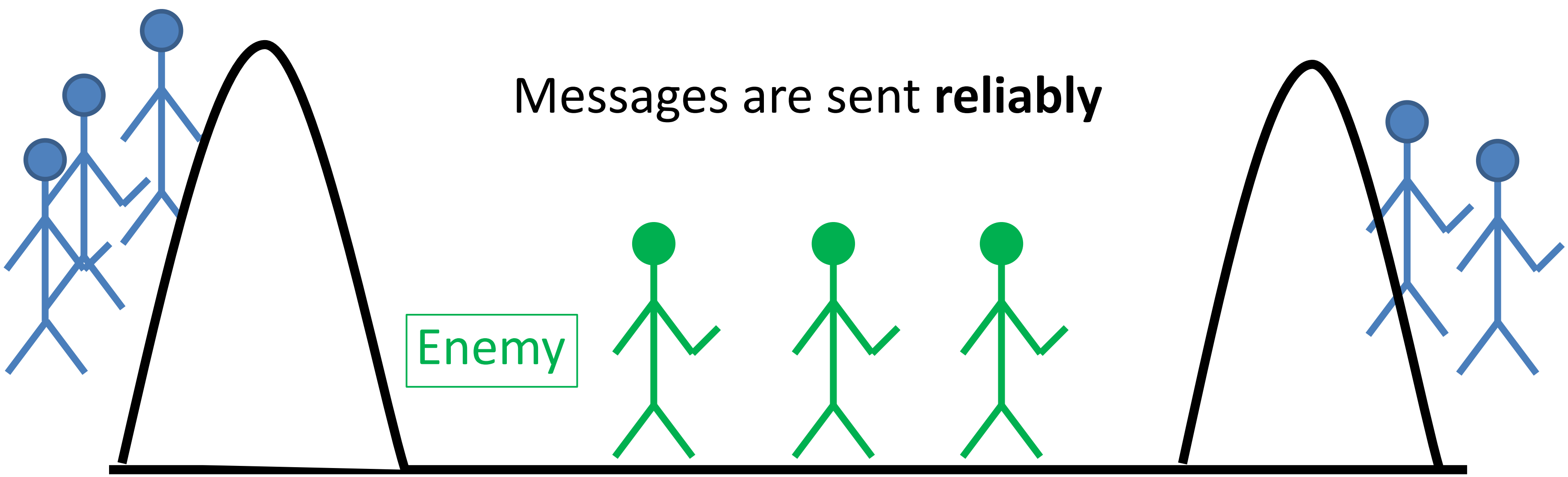
Message takes at least ***min*** time and at most ***max*** time to arrive

Army
Part 1

Attack!

Army
Part 2

Messages are sent **reliably**



Synchronous agreement

When to attack?

Army
Part 1

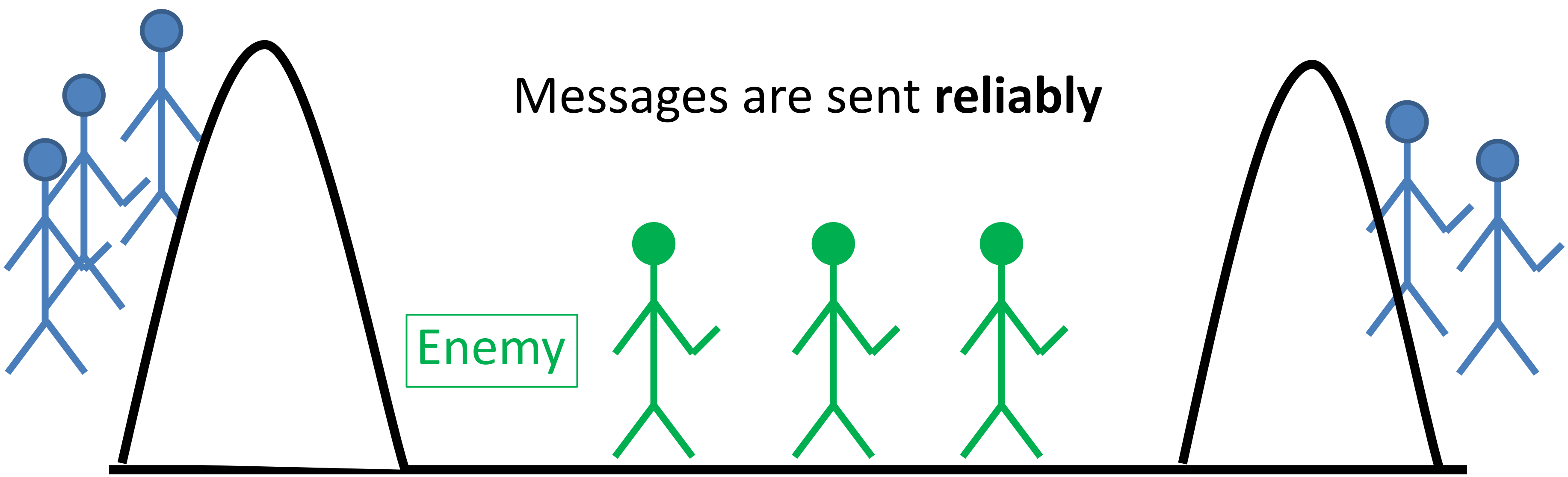
Message takes at least ***min*** time
and at most ***max*** time to arrive

Army
Part 2

Attack!

Strategy: Waits for *min* time then attacks.

Messages are sent **reliably**



Synchronous agreement

When to attack?

Message takes at least ***min*** time
and at most ***max*** time to arrive

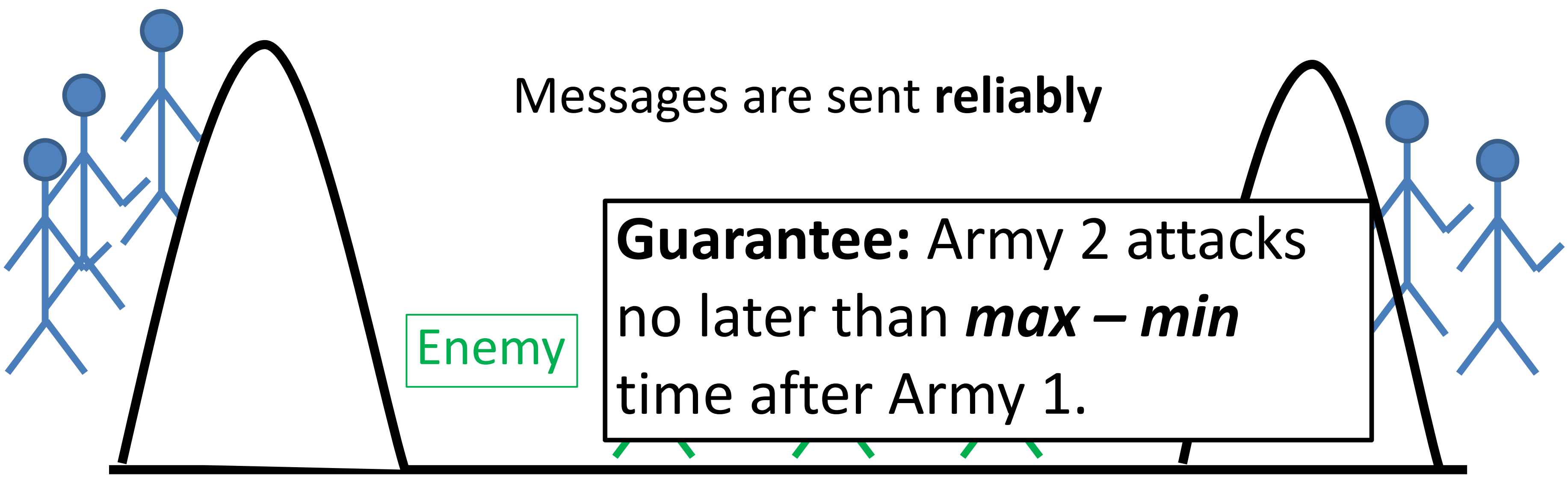
Army
Part 1

Army
Part 2

Attack!

Strategy: Waits for *min* time then attacks.

Messages are sent **reliably**



Guarantee: Army 2 attacks no later than ***max – min*** time after Army 1.

Summarizing takeaways

- Some problems cannot be solved in an asynchronous world (e.g., *when* vs. *who* leads attack)
- A solution valid for asynchronous distributed systems is also valid for synchronous ones (synchronous model is a stronger model)
- Internet and many practical distributed applications are closer to asynchronous than synchronous model
- Apply timeouts and timing assumptions to reduce uncertainty and to bring elements of the synchronous model into the picture

Self-study questions

- Think of a few design problems that cannot be solved in an asynchronous world …
- … and show how they can be solved in a synchronous world.
- What are some other useful assumptions for distributed system design and how practical are they?
- Can you think of other coordination and agreement problems?

You may have come across this before. E.g., ***Operating Systems, Computer Organization, or Computer Architecture***

RECAP: MUTUAL EXCLUSION

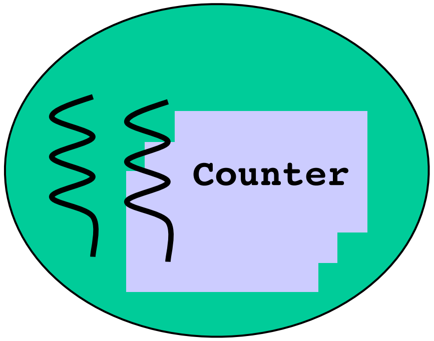
IN A NON-DISTRIBUTED SYSTEMS CONTEXT

Problem: Access to shared variables

- Imagine a **globally shared variable** `counter` accessible to multiple threads (or processes)
  - For example, a **key-value record** managed by a storage server (or more complex data structure)

Let's dissect the issue in detail

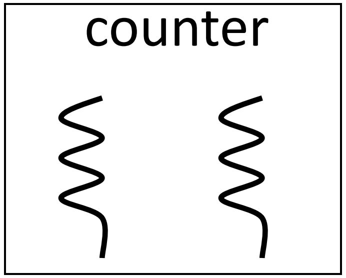
Shared data & synchronization



What may happen if multiple threads concurrently access shared state (e.g., a shared variable)?

Concurrently manipulating shared data

- Two threads execute concurrently as part of the same process
- Shared variable (e.g., global variable)
  - `counter = 5`
- Thread 1 executes
  - `counter++`
- Thread 2 executes
  - `counter--`



- *What are **all the possible values** of* `counter` *after Thread 1 and Thread 2 executed?*

Machine-level implementation

- Implementation of “`counter++`”

  $register_1$ = counter

  $register_1$ = $register_1$ + 1

  counter = $register_1$

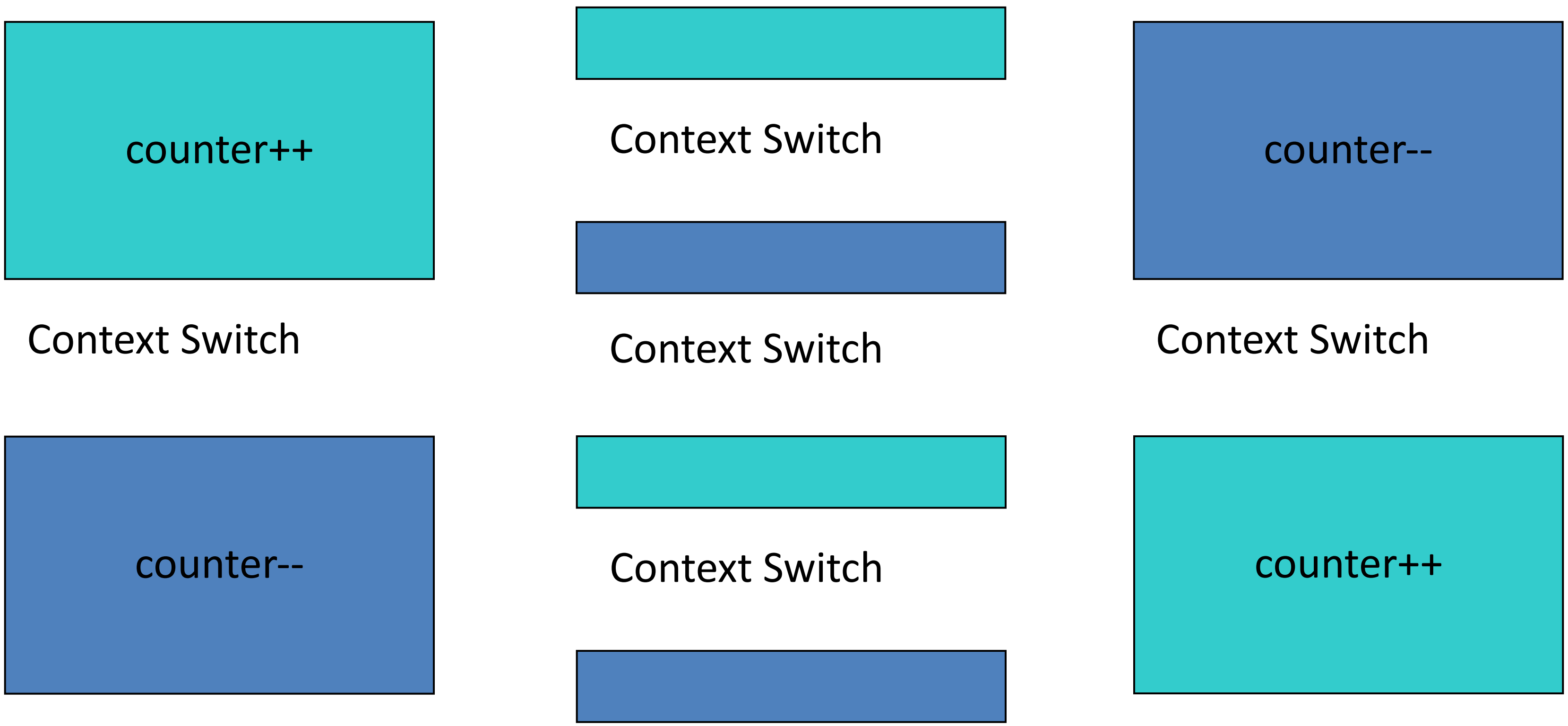
- Implementation of “`counter--`”

  $register_2$ = counter

  $register_2$ = $register_2$ – 1

  counter= $register_2$

Possible execution sequences



Interleaved execution

- Assume **counter** is 5 and interleaved execution of counter++ (in T_1) and counter-- (in T_2)

T_1: r_1 = counter ($register_1 = 5$)

T_1: r_1 = $r_1 + 1$ (***$register_1 = 6$***)

T_2 : r_2 = counter ($register_2 = 5$)

T_2 : r_2 = $r_2 - 1$ (***$register_2 = 4$***)

T_1 : **counter = r_1** (***counter = 6***)

T_2 : counter = r_2 (***counter = 4***)

Context switch

- The value of **counter** may be 4 or 6, whereas **the correct result should be 5**!

Race condition

- **Race condition**
  - **Several** threads **manipulate shared data concurrently**. The **final value** of the data **depends** upon **which thread finishes last**.
- In our example (interleaved execution) of `counter++` with `counter--`
- To prevent race conditions, concurrent processes must be **synchronized**!

The moral of this story

- The statements

  ```
  counter++;
  counter--;
  ```

  must each be executed ***atomically***.
- Atomic operation means an operation that **completes in its entirety without interruption**.
- This is achieved through **synchronization primitives**
- Shared variable accessed in a **critical section** must be protected by synchronization primitives
- Known as the **critical section problem** or as **mutual exclusion**

Self-study questions

- Do we have a critical section problem in distributed systems? – There is no shared memory!
- Asked differently, do we need to worry about (distributed) mutual exclusion in a distributed system?
- Identify a few mutual exclusion scenarios in distributed systems.

"*Don't worry*", in distributed systems (the ones we look at), there is no shared memory – but, …

DISTRIBUTED MUTUAL EXCLUSION

OVERVIEW

Distributed mutual exclusion

- In distributed systems, mutual exclusion is at least equally complex due:
  - Lack of shared memory
  - Lack of a global clock
  - Event ordering
- Examples
  - Accessing a shared resource in distributed systems
  - Acquiring a lock
  - One active master to coordinate activities

Critical section problem

No shared memory

- System with *n* nodes
- Nodes access shared resources in CS
- **Coordinate access to CS via message passing**
- Application-level protocol for accessing CS
  - Enter_CS() – enter CS, block if necessary
  - ResourceAccess() – access shared resource in CS
  - Exit_CS() – leave CS

Assumptions

No practical rather theoretical considerations

- System is asynchronous
  - No bound on delays, no bound on clock drift, etc.
- Nodes do not fail
- Message delivery is reliable
  - Any message sent, is eventually delivered intact and exactly once – i.e., not lost, not duplicated
- Nodes are well-behaved and spent finite time accessing resources in CS

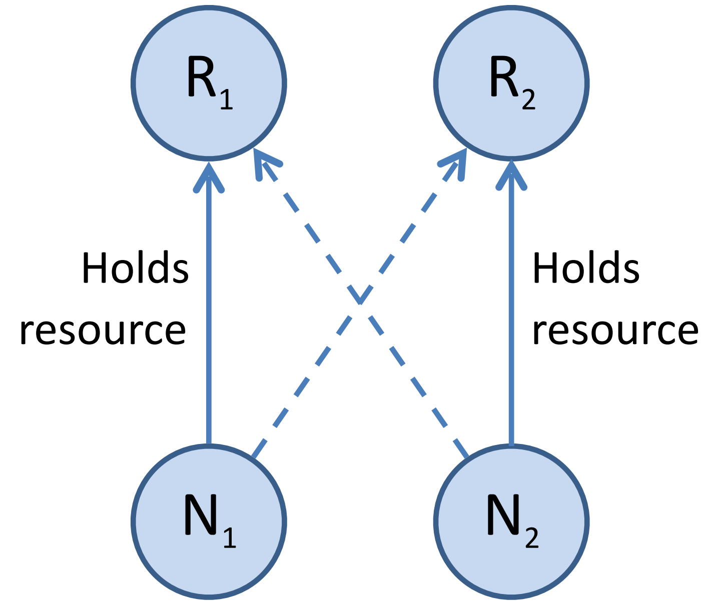
Mutual exclusion requirements

- **Safety** – correctness
  - At most one node in the critical section at a time
- **Liveness –** progress (something good happens)
  - Requests to enter/exit CS eventually succeed
  - No deadlock
- **Fairness** (order & starvation)
  - If one request to enter CS **happened-before** another one, then entry to CS is granted in that **order**
  - Requests are ordered such that no node enters the critical section twice while another waits to enter (i.e., **no starvation**)

Deadlock & starvation

- **Deadlock**: Two or more nodes become **stuck indefinitely** while attempting to enter and exit CS – e.g., by virtue of their mutual dependency

- **Starvation**: The **indefinite postponement** of entry to CS for a node that has requested it.



Mutually wait for resources held by another node

Possible performance metrics

- **Bandwidth:** Number of messages sent, received or both
- **Synchronization delay**: Time between one process **exiting** critical section and next one **entering**
- **Client delay**: Delay **at entry** and **exit** (response time)

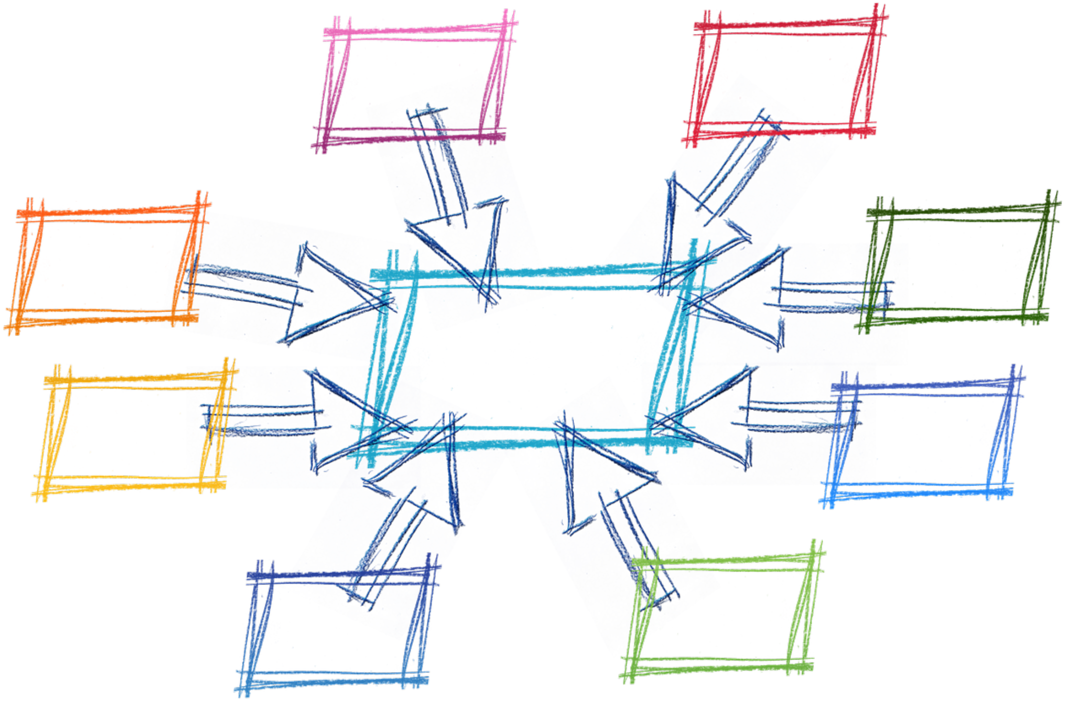
- We do not measure client access to resources protected by the critical section (**assume finite**)

Solution strategy overview

- **Centralized strategy**
  - Divide nodes into **leader** and **follower**, leader dictates actions of followers
- **Distributed strategy**: Each node independently decides actions, based on local knowledge of others' state
  - **Token-based**: A node is allowed in the critical section (CS) if it has a token. Tokens are passed from node to node, in some (priority) order.
  - **Non-token-based**: A node enters CS when an **assertion becomes true**. A node communicates with other nodes to obtain their states and decides whether the assertion is true or false.

Self-study questions

- What are some examples for mutual exclusion scenarios in distributed systems?
- What are some scenarios where we need locks in distributed systems, after all, there is no shared memory (in our notions of DS)



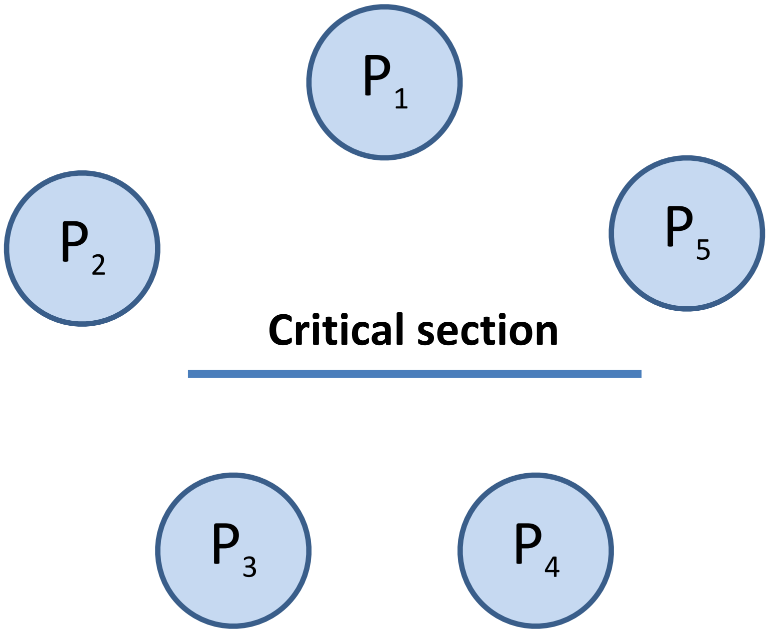
Pixabay license

DISTRIBUTED MUTUAL EXCLUSION

CENTRALIZED STRATEGY

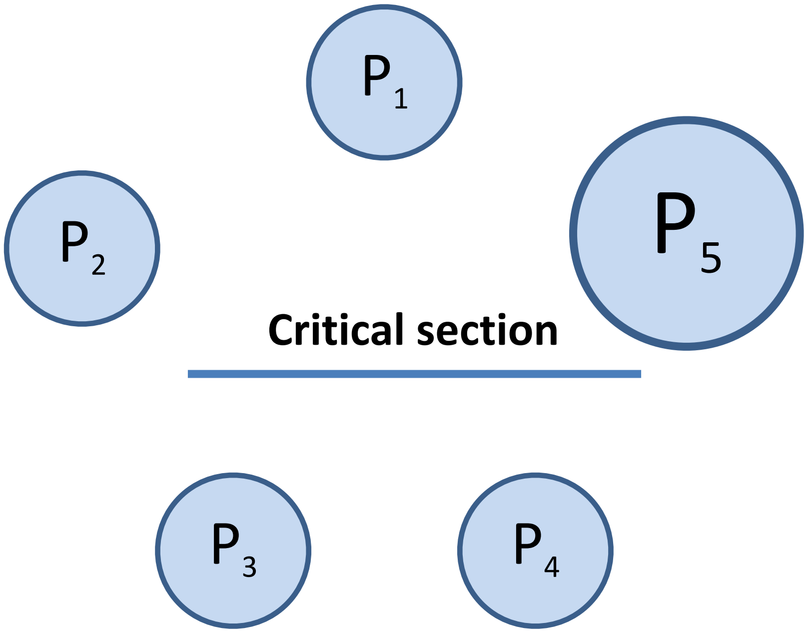
Centralized strategy: Elect leader

Elect a leader (not in scope of this strategy)



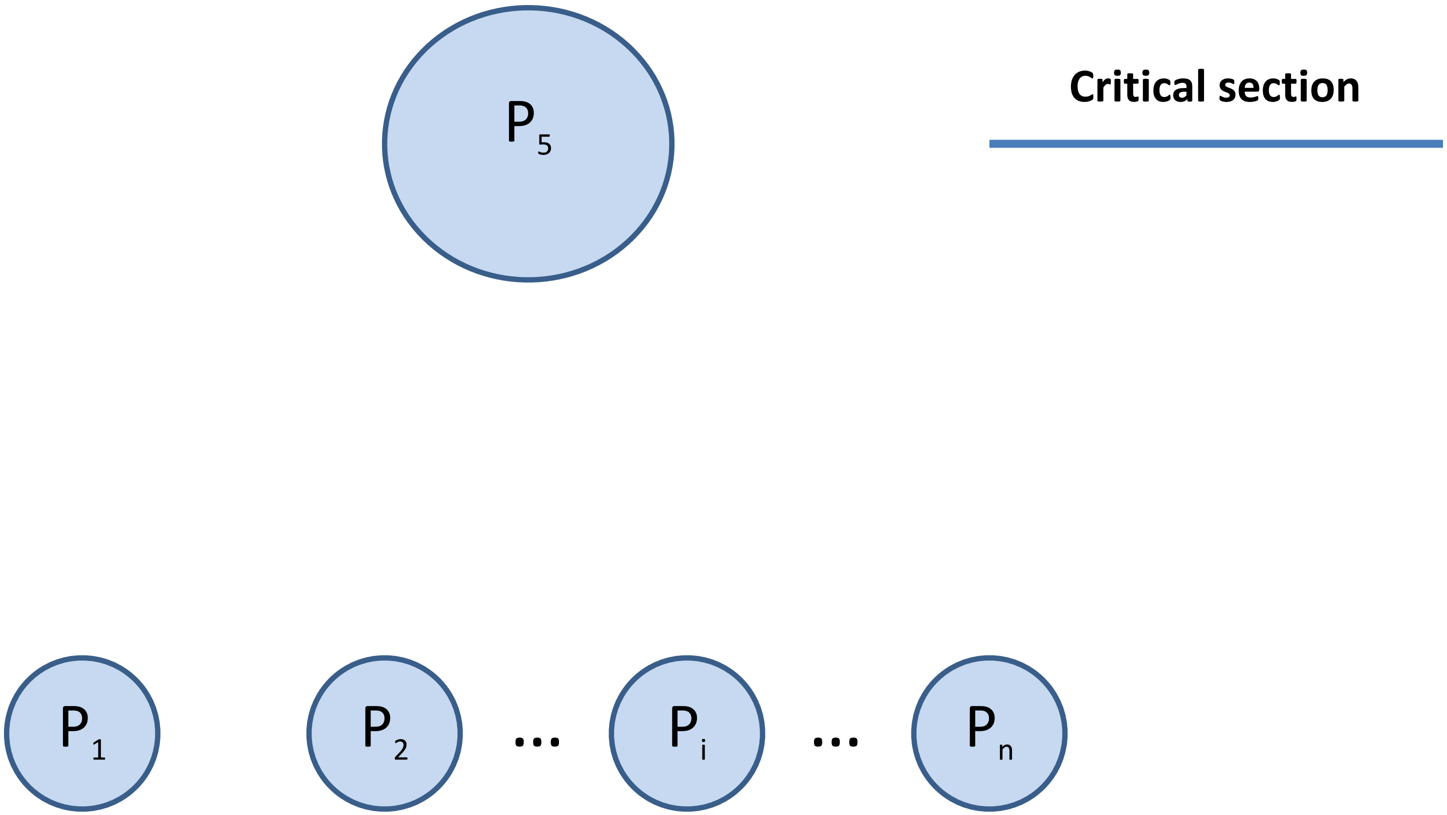
Centralized strategy: Elect leader

Elect a leader (not in scope of this strategy)



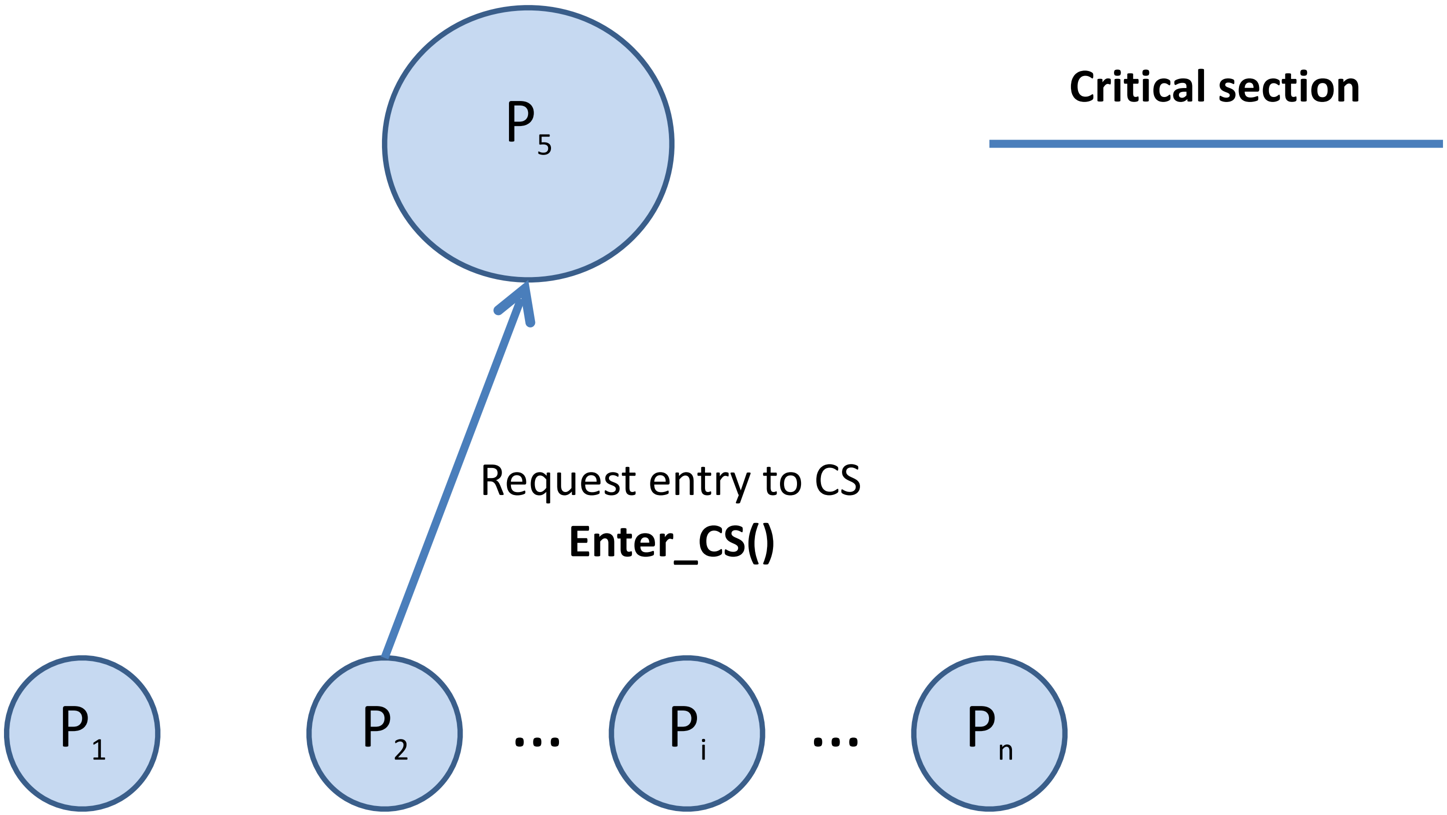
Centralized strategy: Empty CS

A.k.a. server / coordinator / **leader** / master



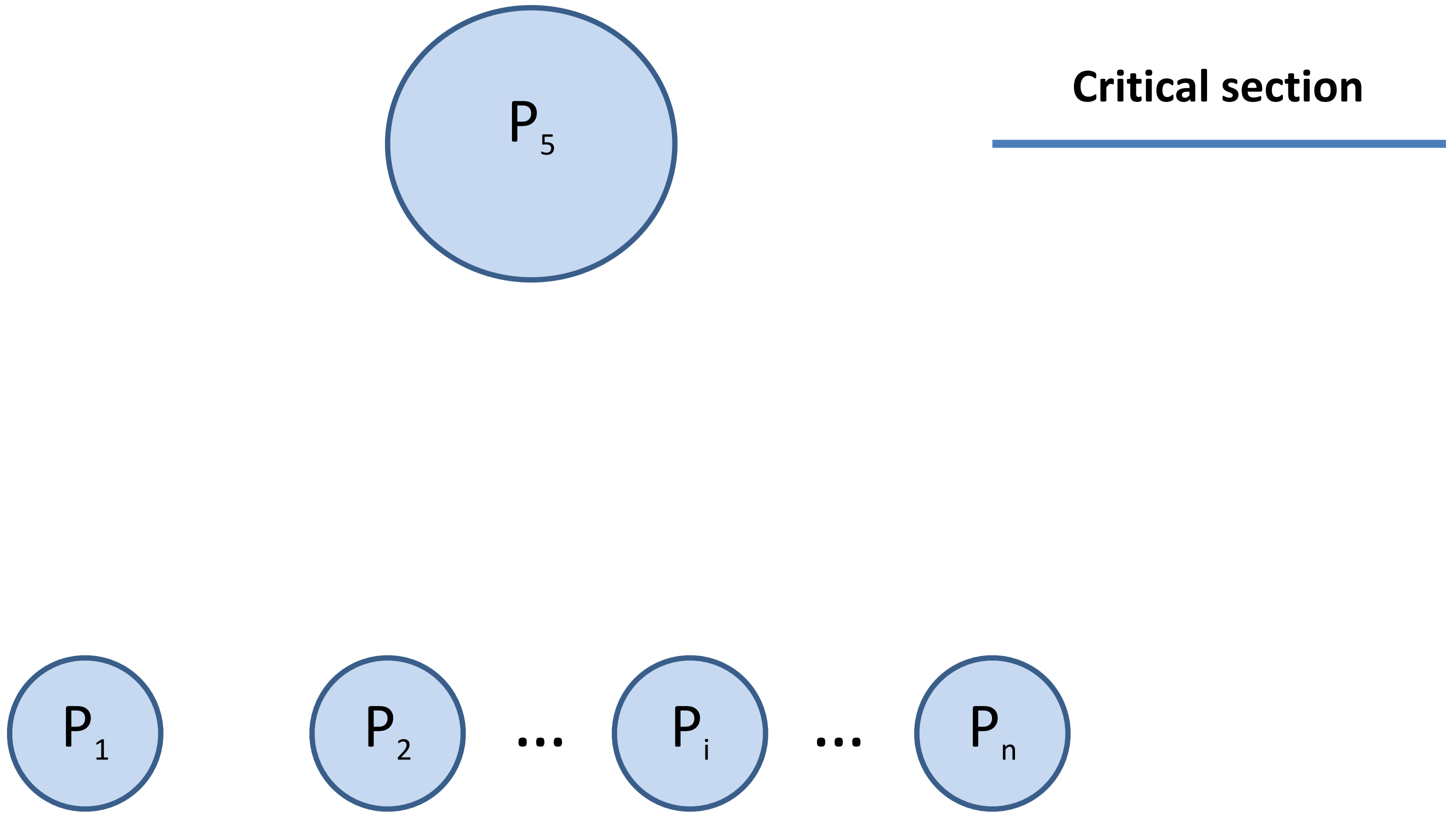
Centralized strategy: Empty CS

A.k.a. server / coordinator / **leader** / master



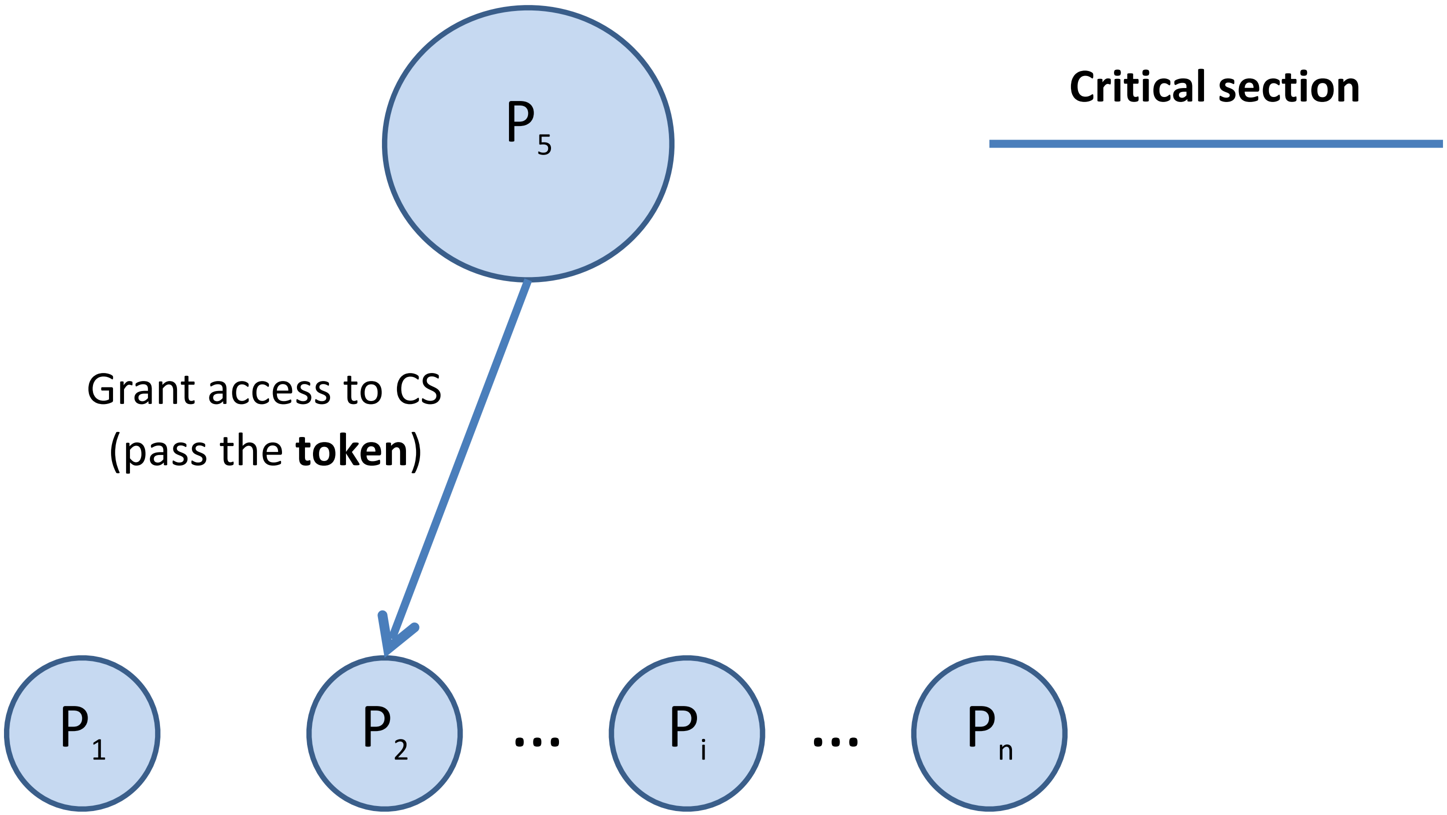
Centralized strategy: Empty CS

A.k.a. server / coordinator / **leader** / master



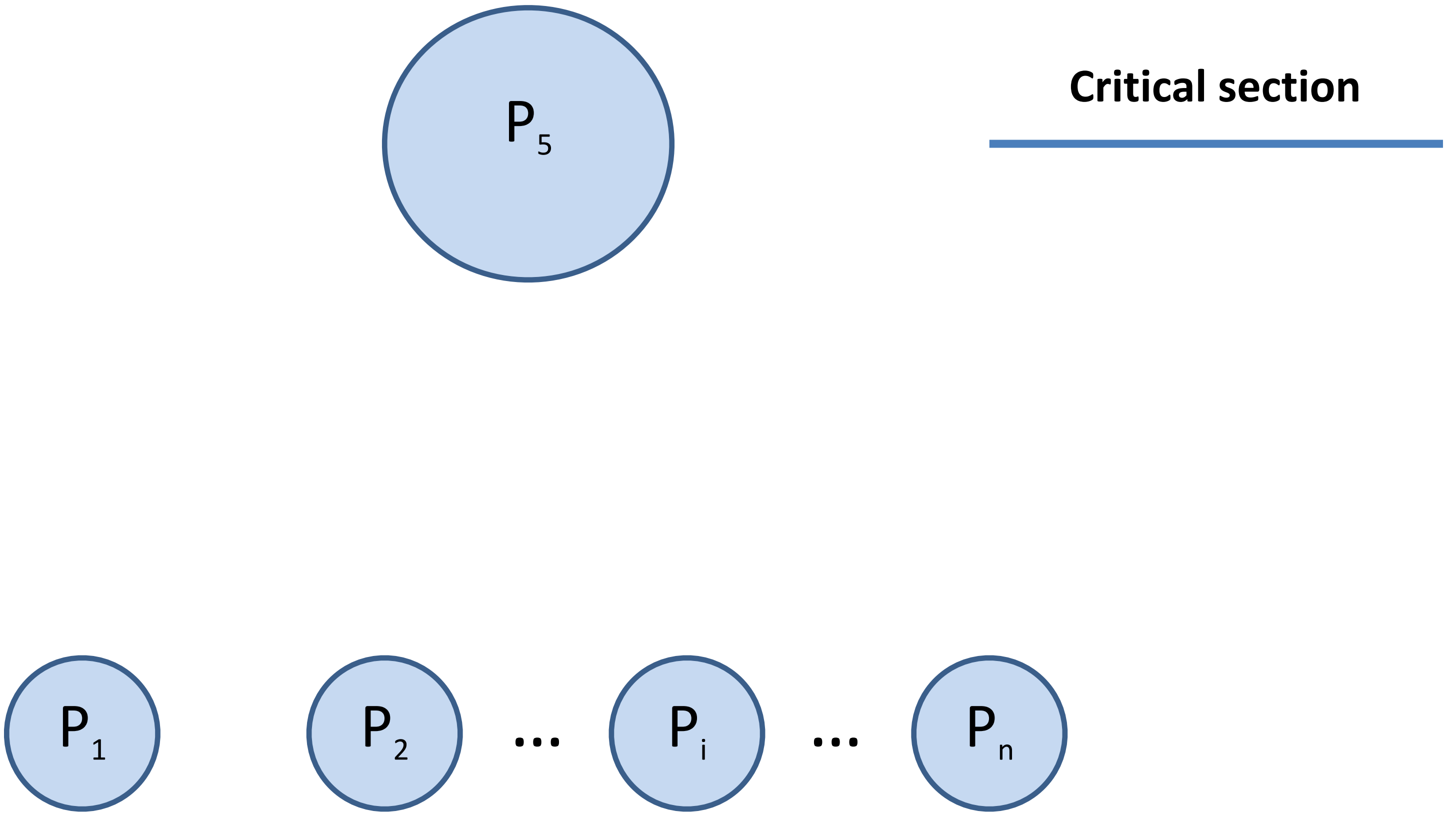
Centralized strategy: Empty CS

A.k.a. server / coordinator / **leader** / master



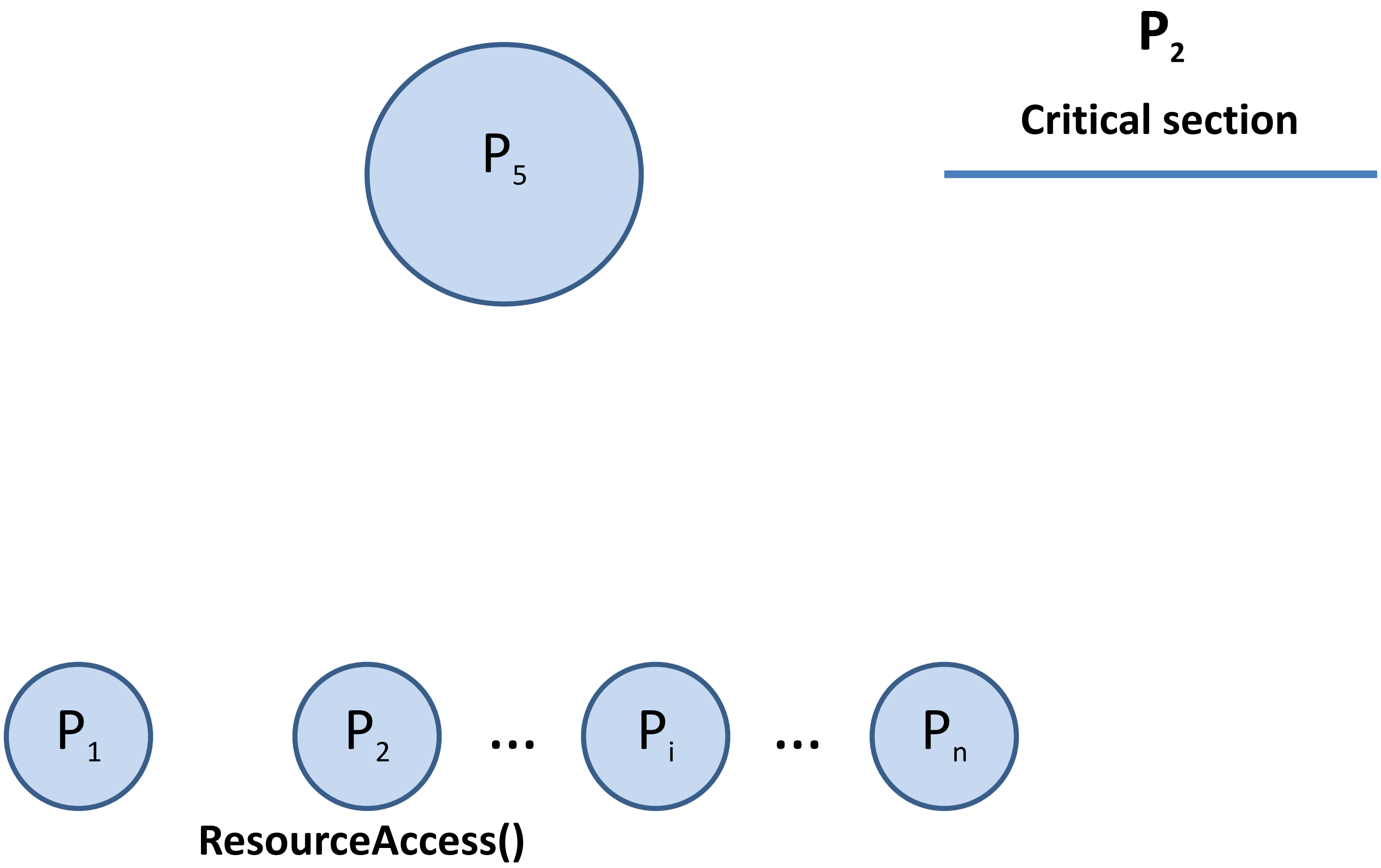
Centralized strategy: Empty CS

A.k.a. server / coordinator / **leader** / master



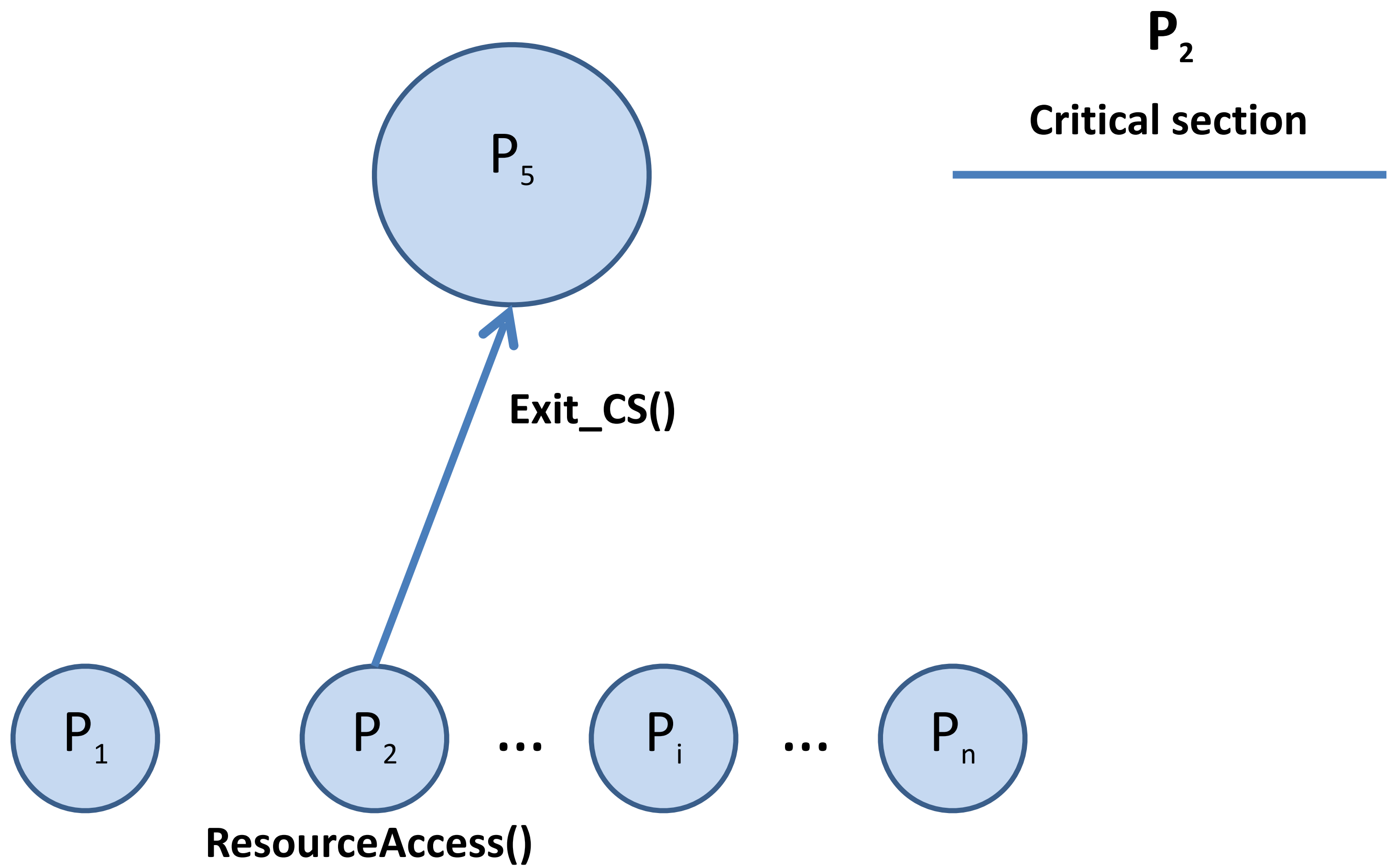
Centralized strategy: Empty CS

A.k.a. server / coordinator / **leader** / master



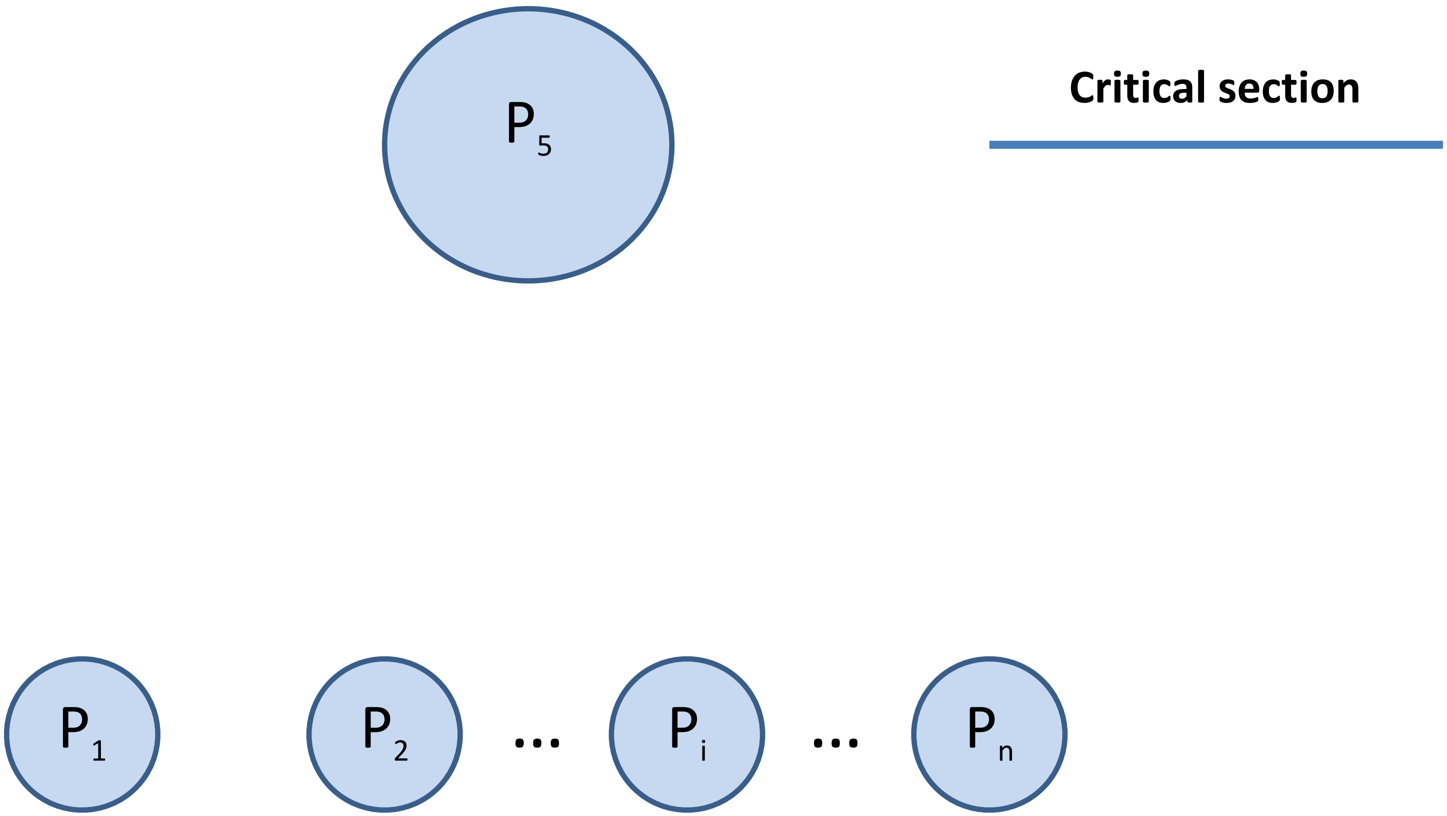
Centralized strategy: Empty CS

A.k.a. server / coordinator / **leader** / master

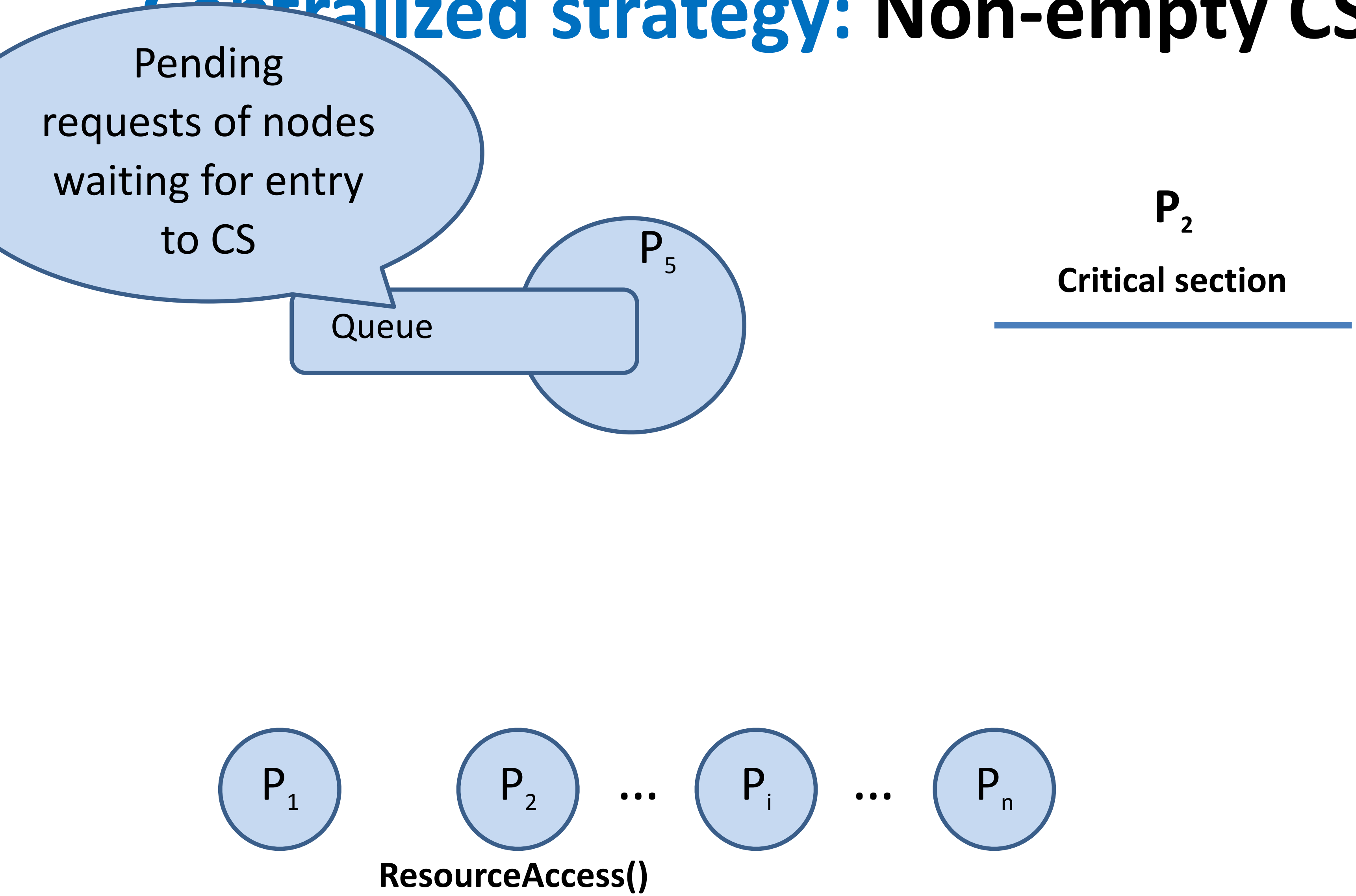


Centralized strategy: Empty CS

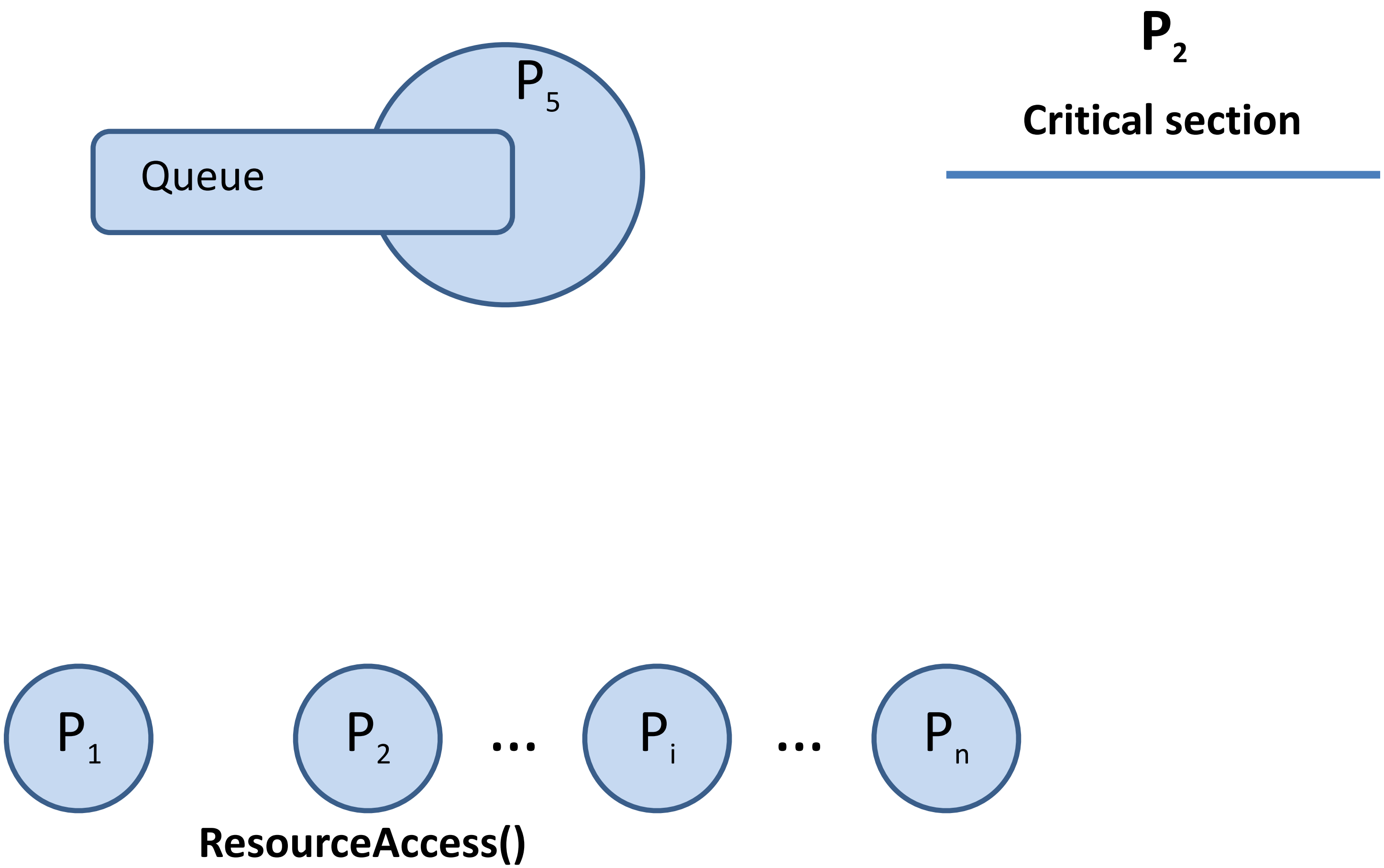
A.k.a. server / coordinator / **leader** / master



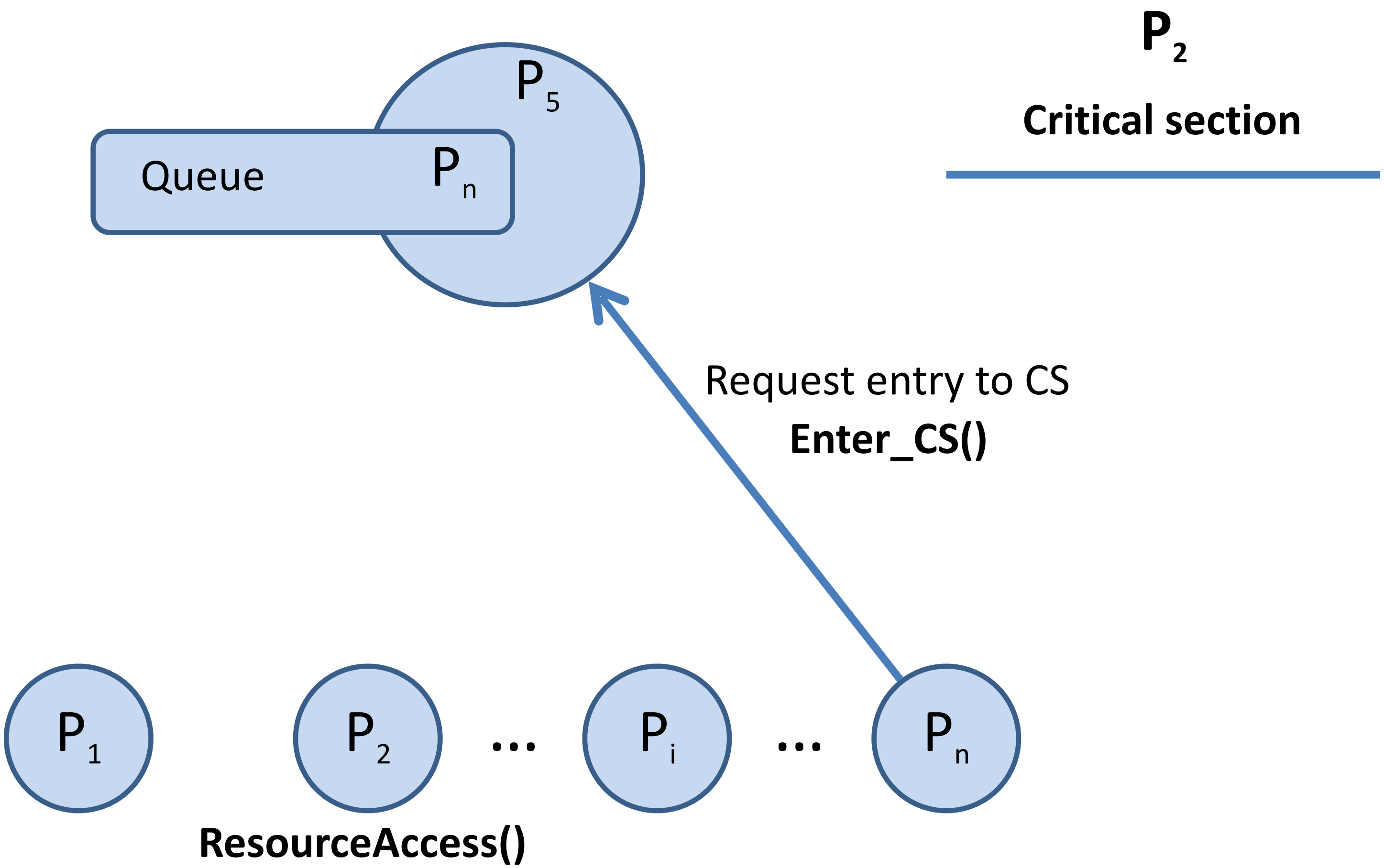
Centralized strategy: Non-empty CS



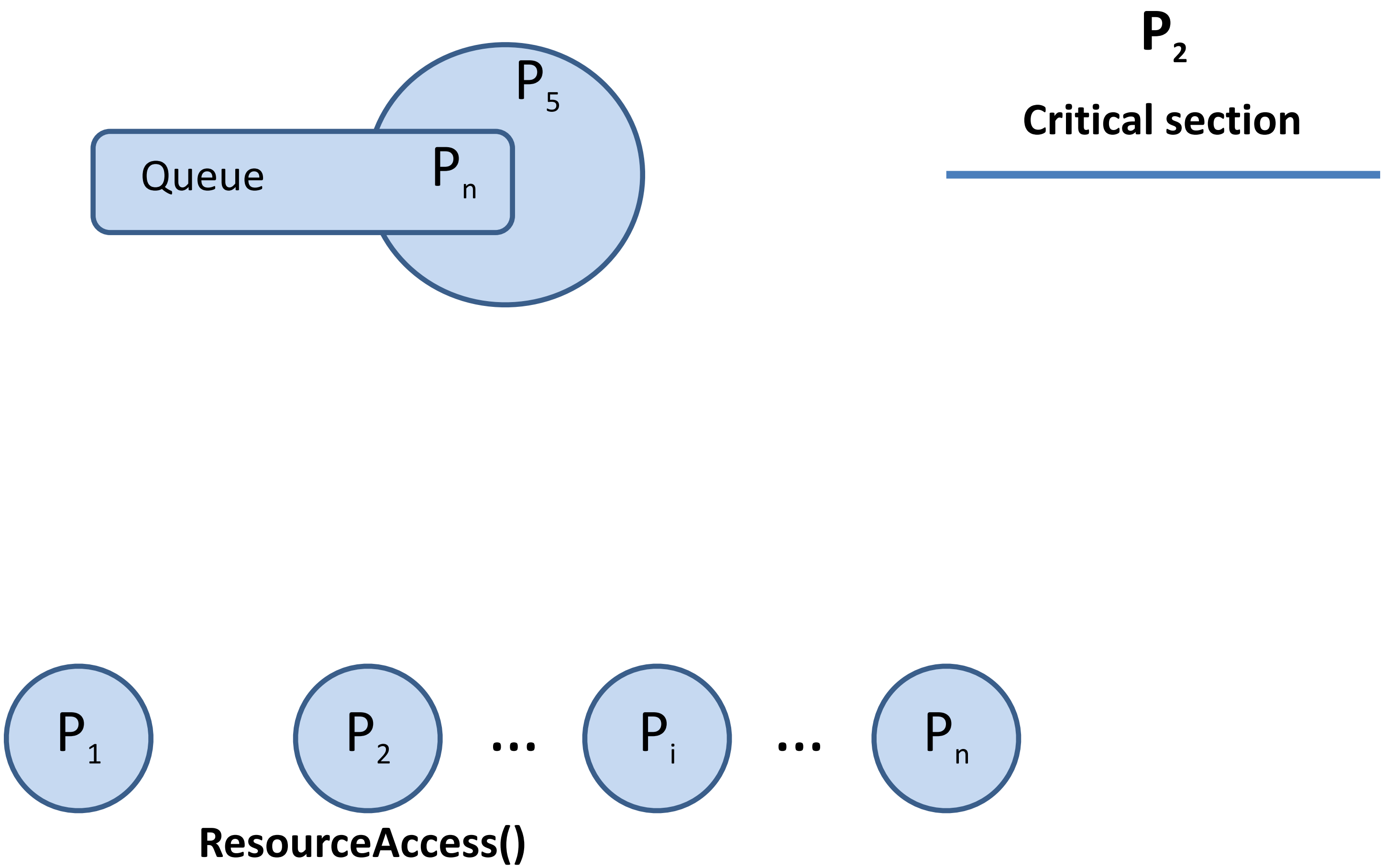
Centralized strategy: Non-empty CS



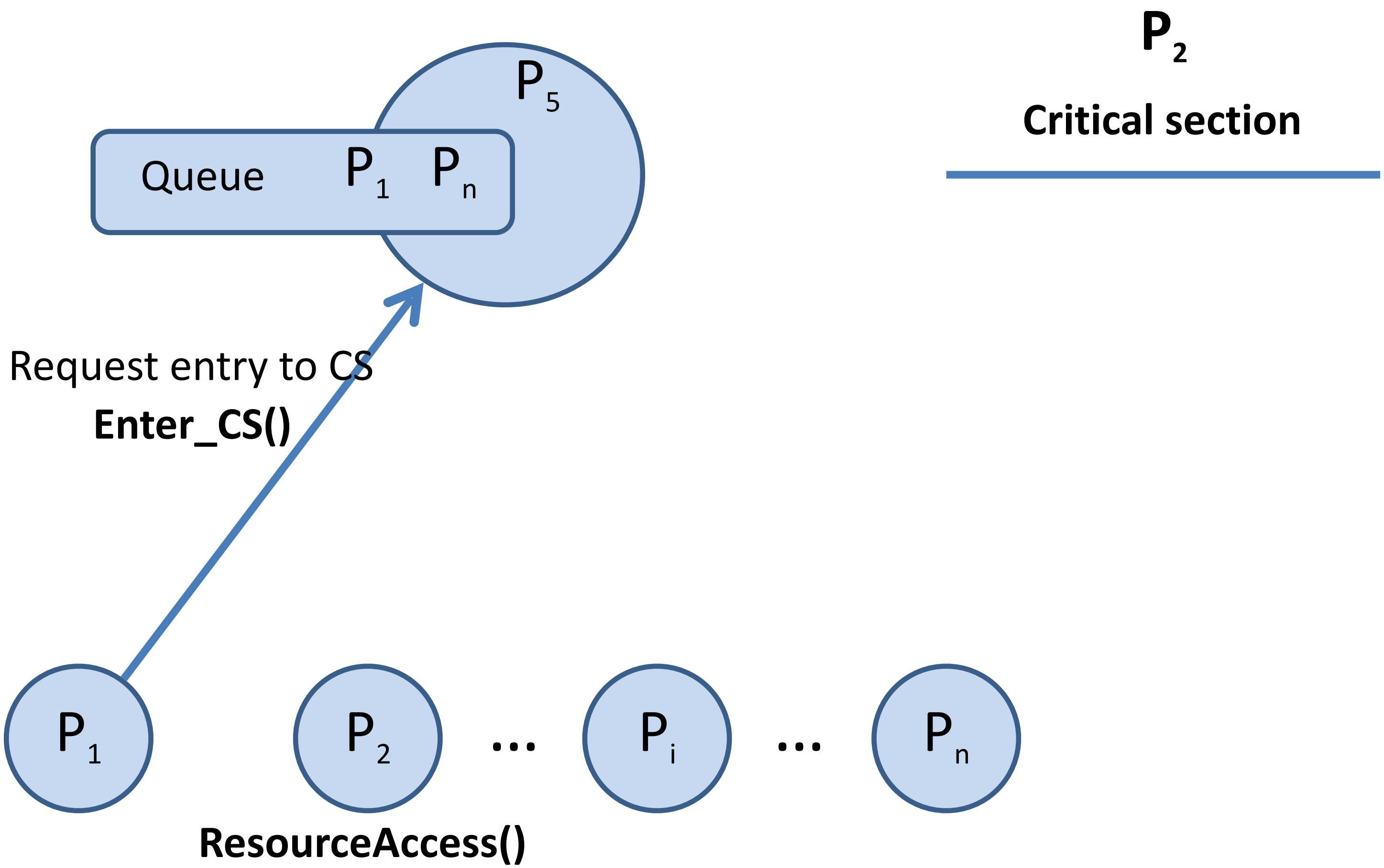
Centralized strategy: Non-empty CS



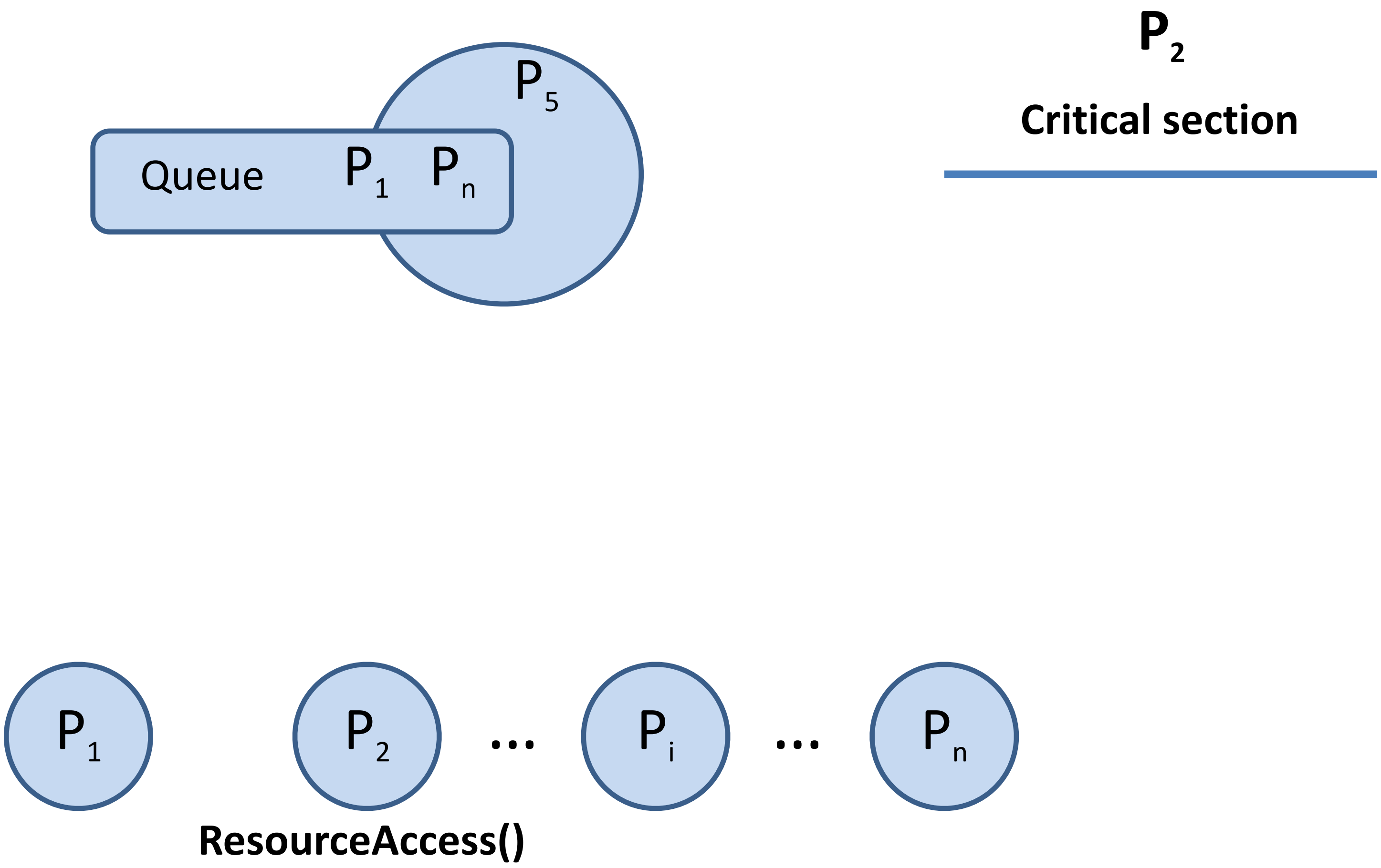
Centralized strategy: Non-empty CS



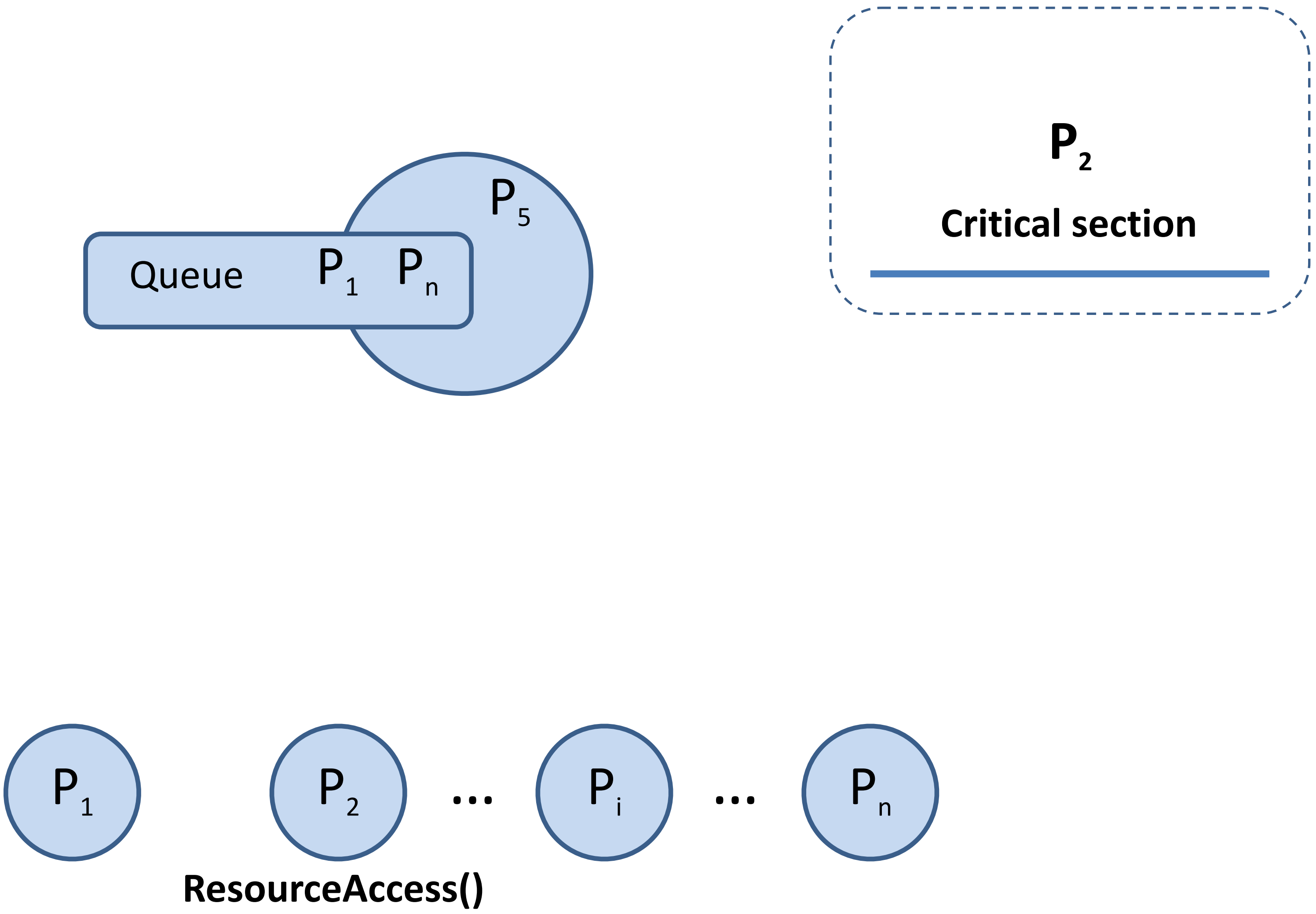
Centralized strategy: Non-empty CS



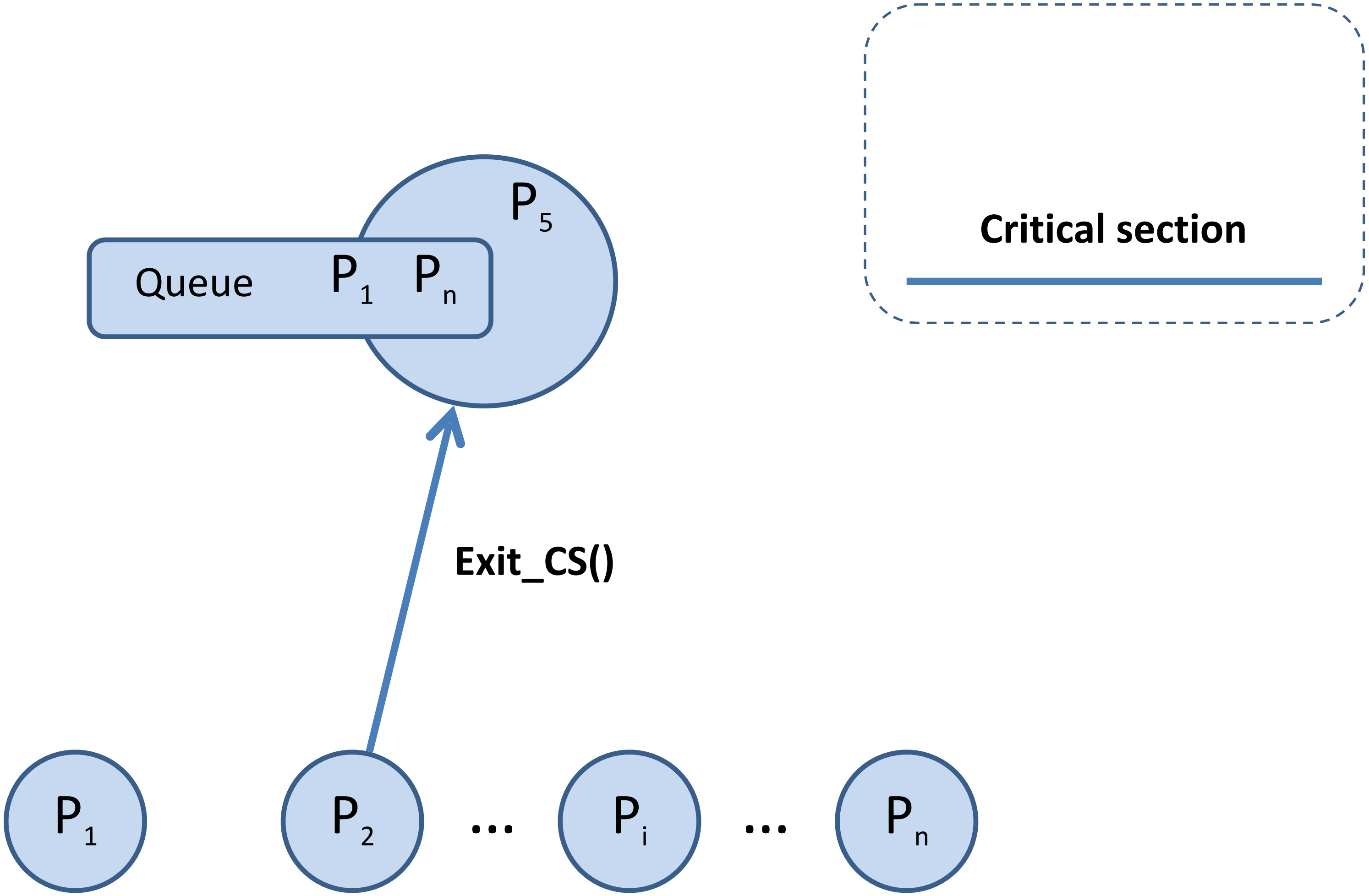
Centralized strategy: Non-empty CS



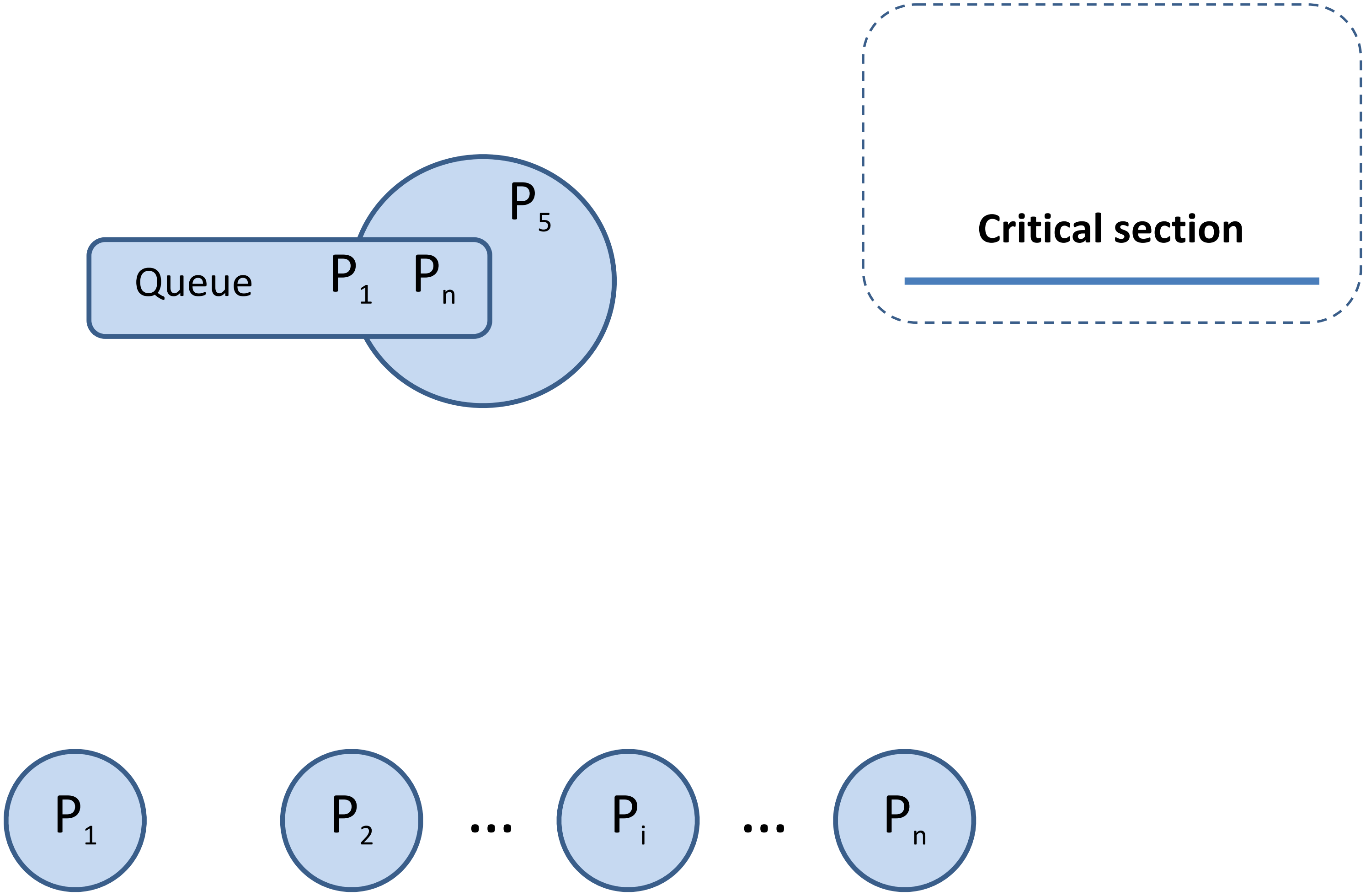
Centralized strategy: Exit CS



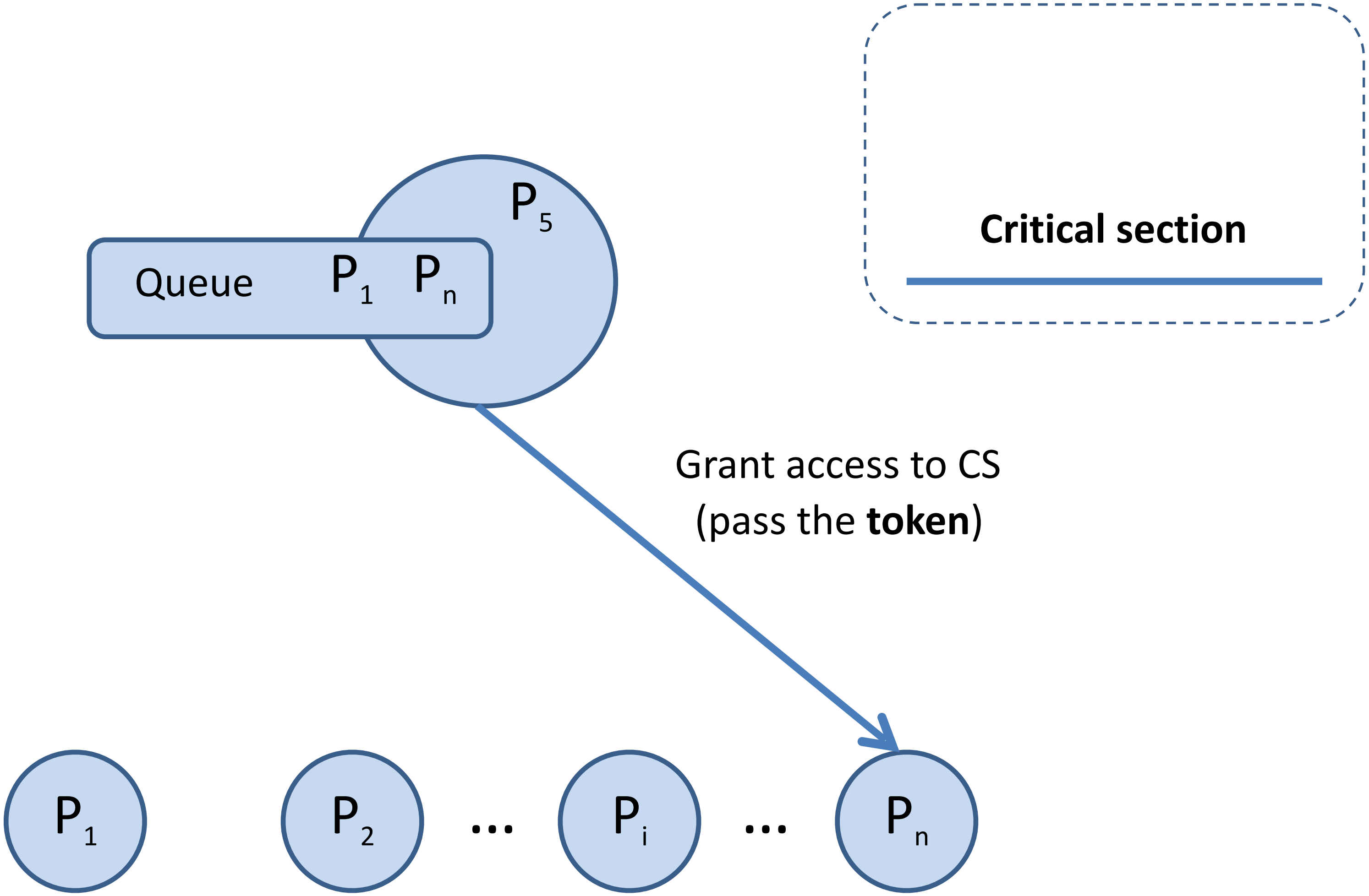
Centralized strategy: Exit CS



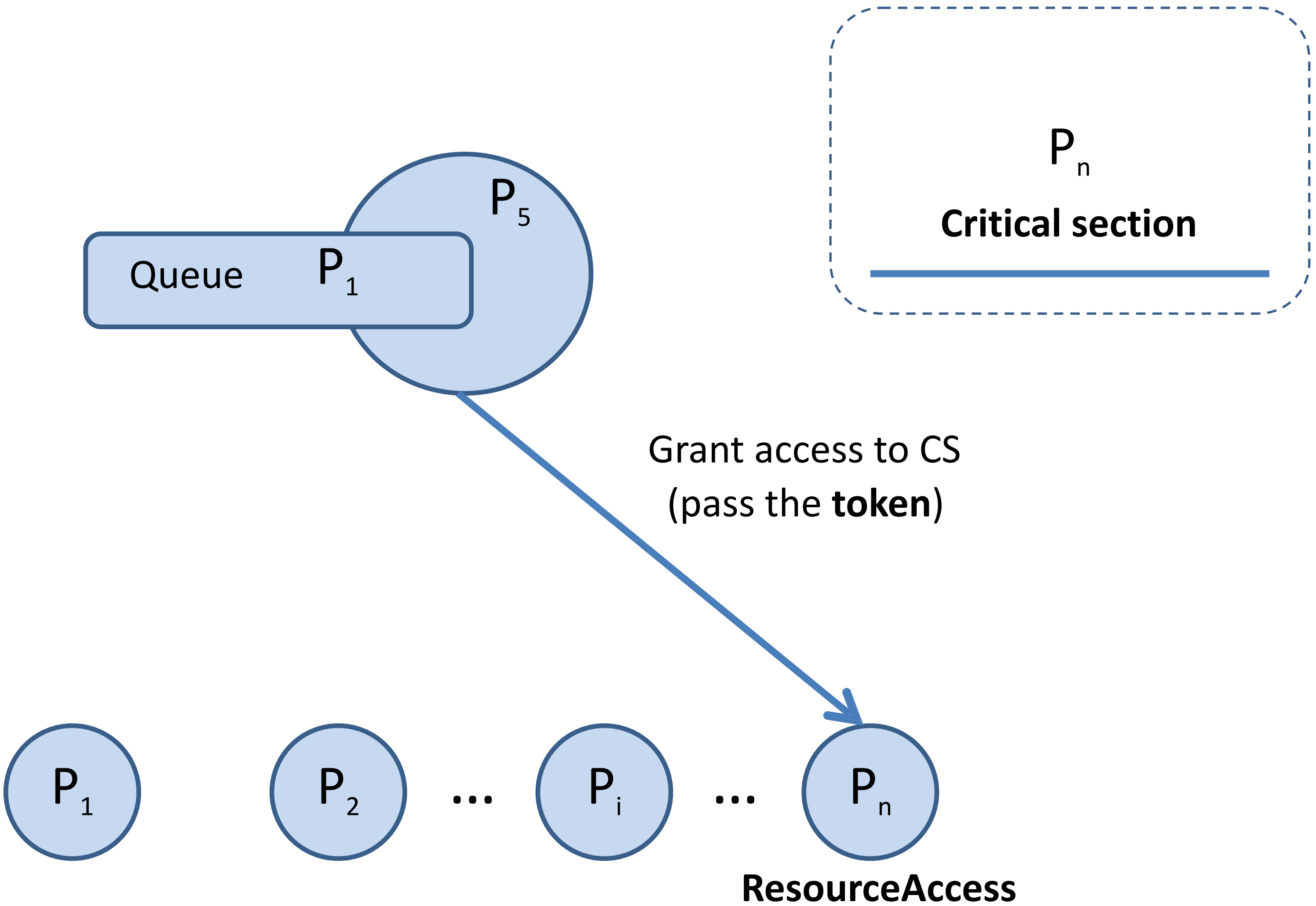
Centralized strategy: Exit CS



Centralized strategy: Exit CS



Centralized strategy: Exit CS



Summarizing observations I

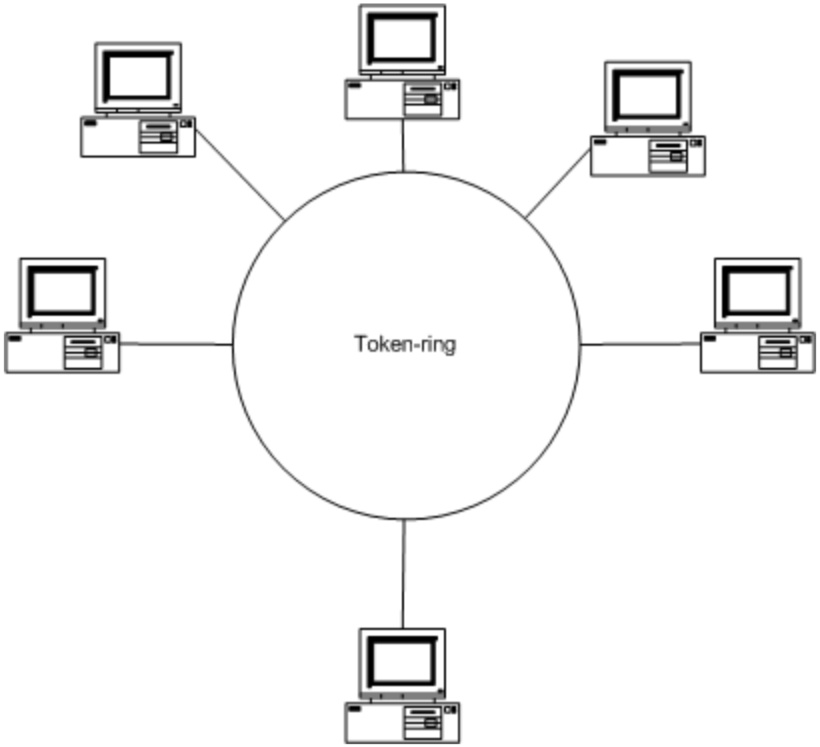
- Meets requirements: Safety, liveness, no starvation
- ***Does solution meet the ordering requirement?***
- Advantages
  - **Simple to implement**
- Disadvantages
  - **Single point of failure**
  - Bottleneck, network congestion, timeout
- Deadlock potential for multiple resources with separate servers

Summarizing observations II

- Enter_CS()
  - **Two messages**: Request & Grant
  - One round of communication (RTT delay)
- Exit _CS()
  - **One message**: Release message
  - No delay for the node in CS

Self-study questions

- Does solution meet the ordering requirement? Why or why not?
- How could a single leader manage multiple resources? Analyse pros and cons of alternative designs?
- Provide a differentiated discussion regarding failure of leader, follower, follower in CS, follower requested access, etc.)

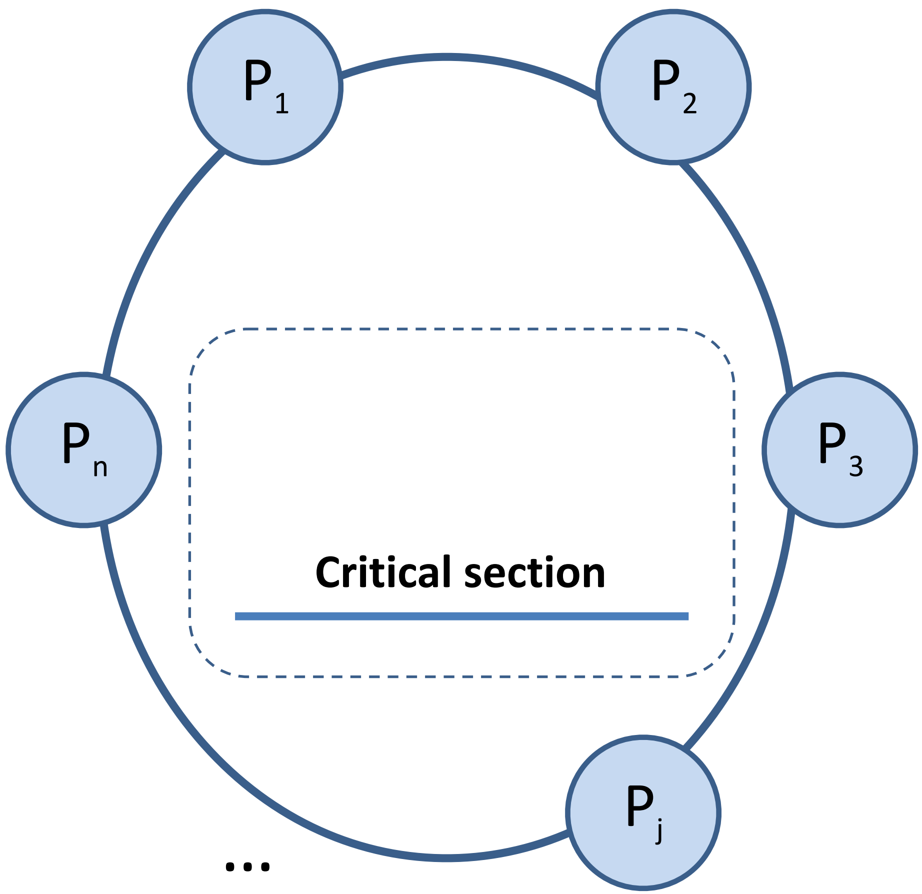


DISTRIBUTED MUTUAL EXCLUSION

RING-BASED ALGORITHM

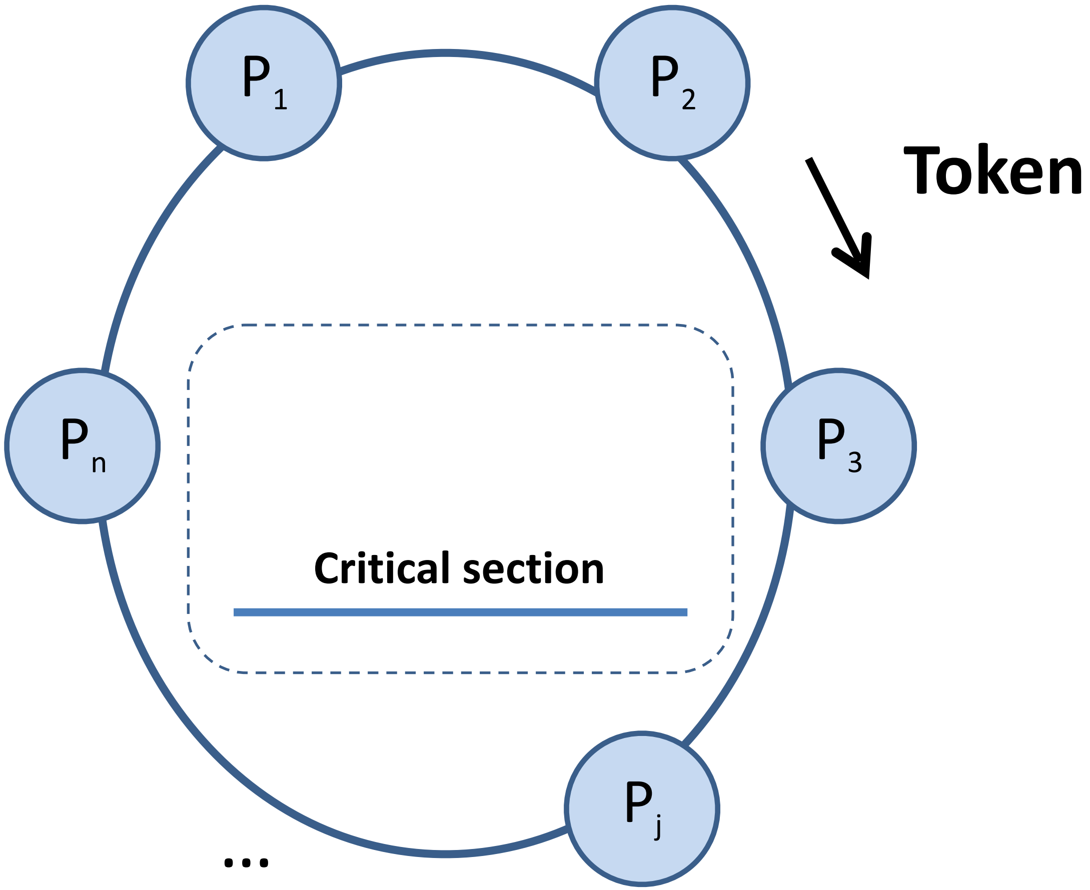
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



Ring-based algorithm

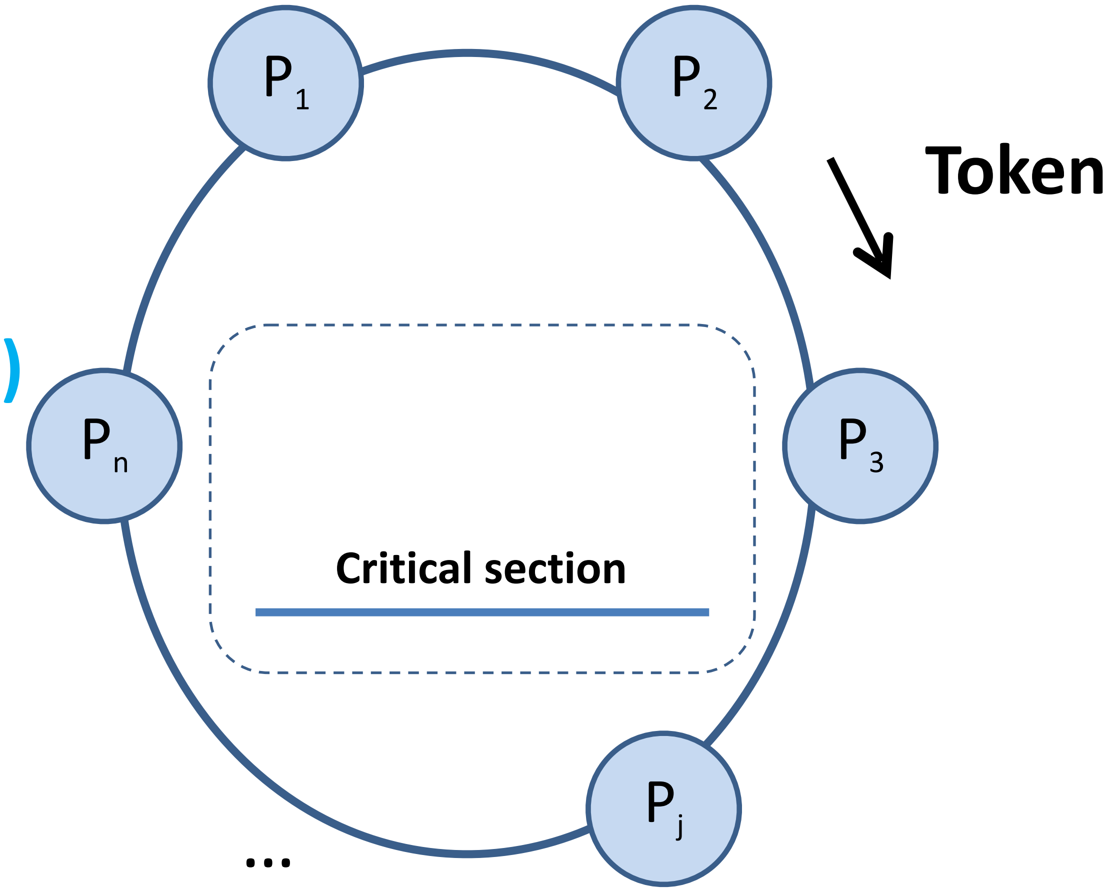
- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



Ring-based algorithm

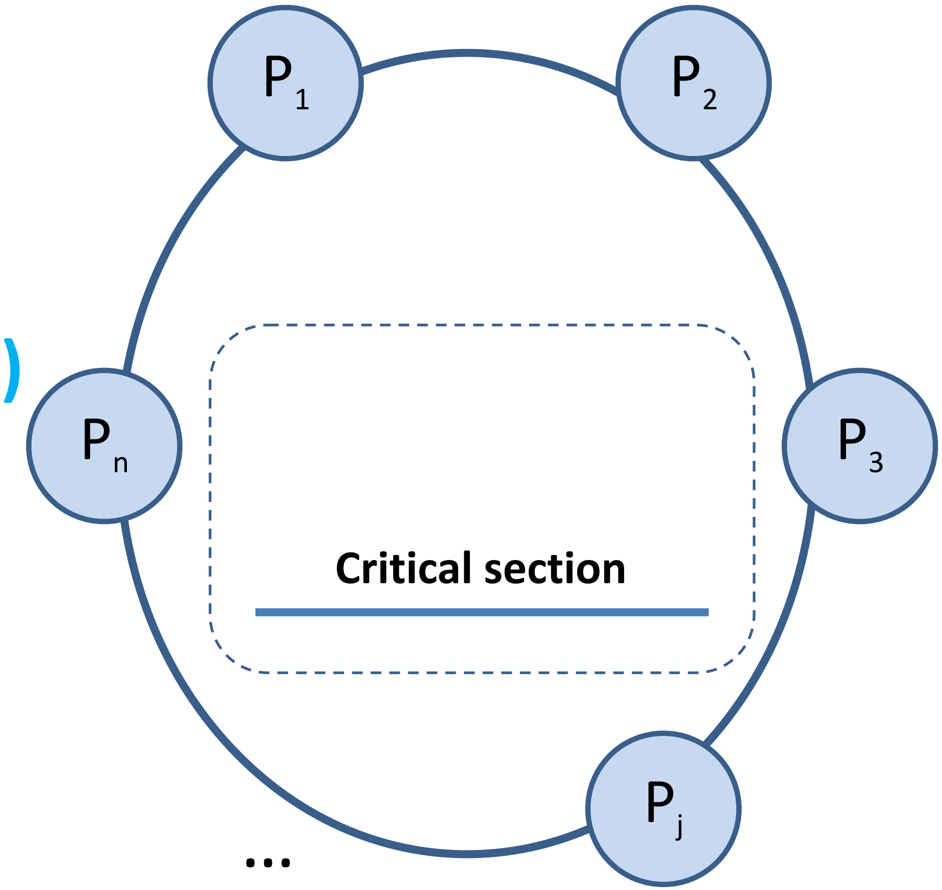
- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology

Enter_CS()



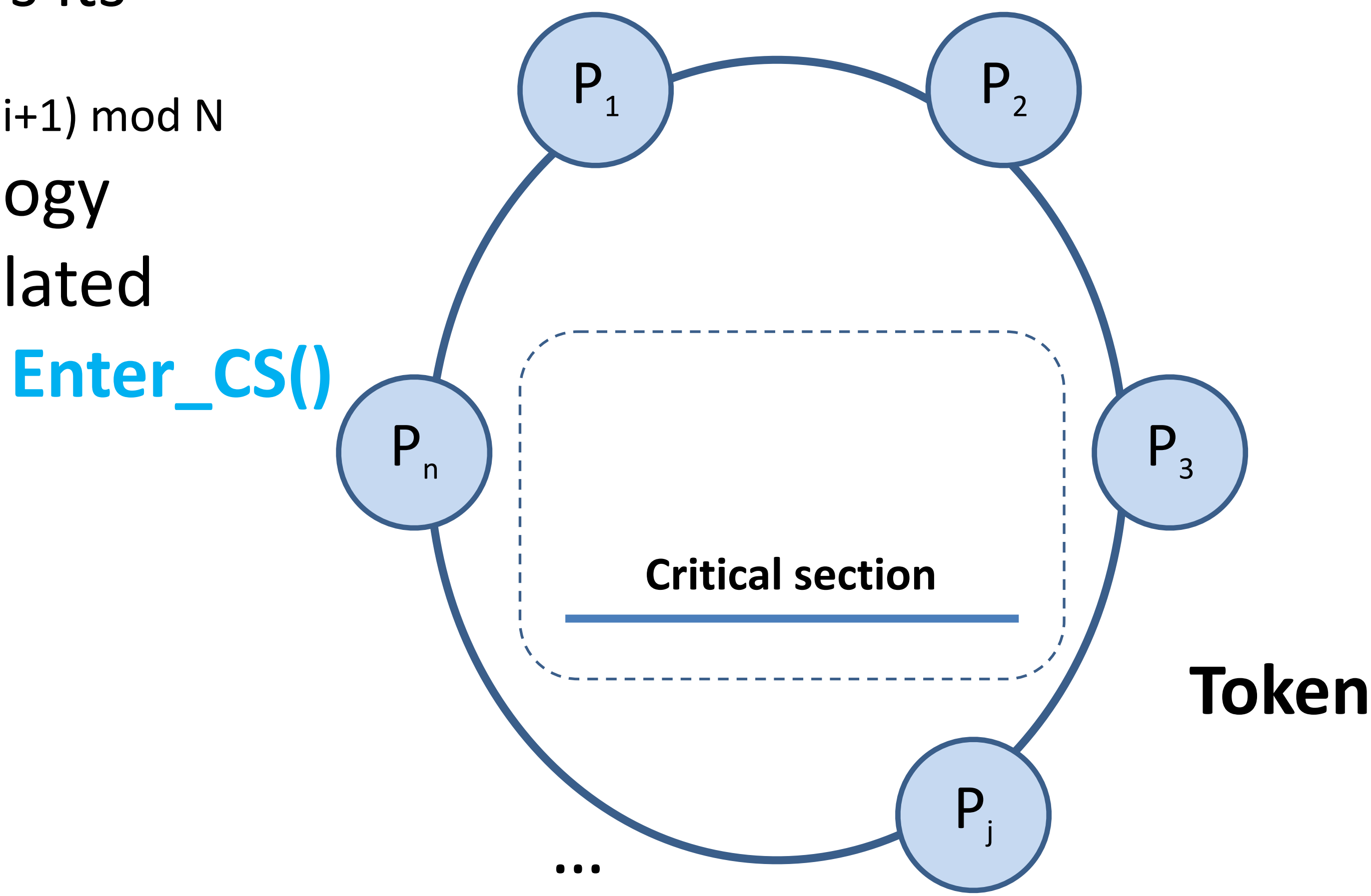
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



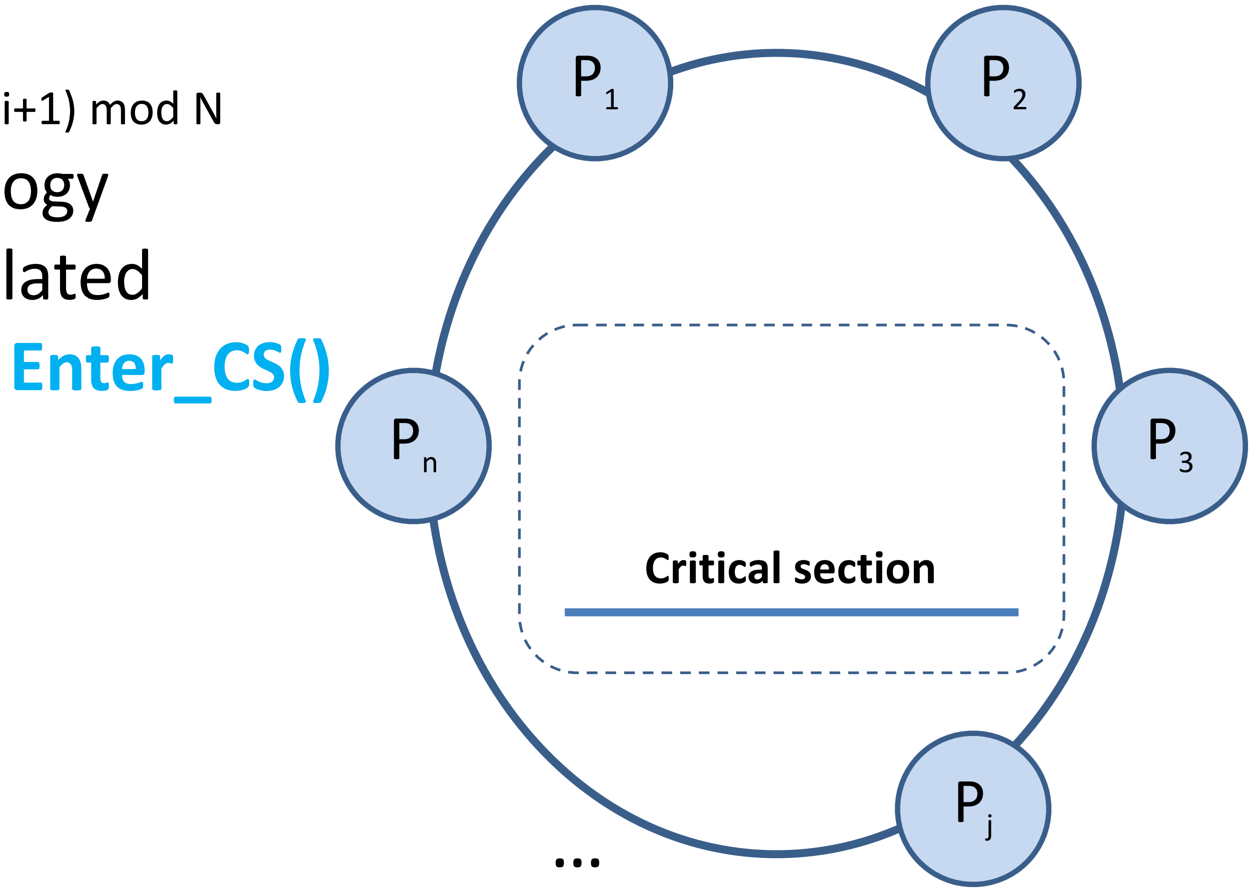
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



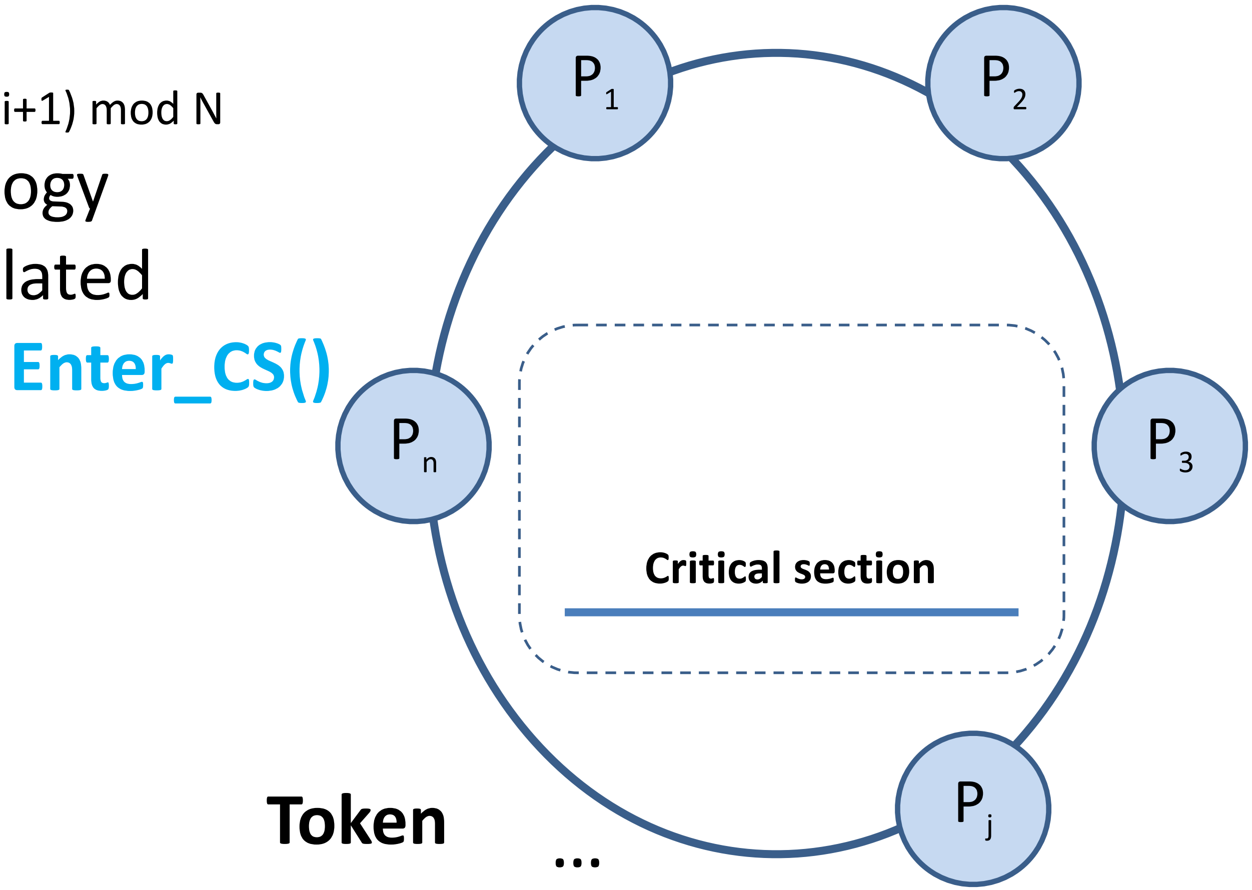
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



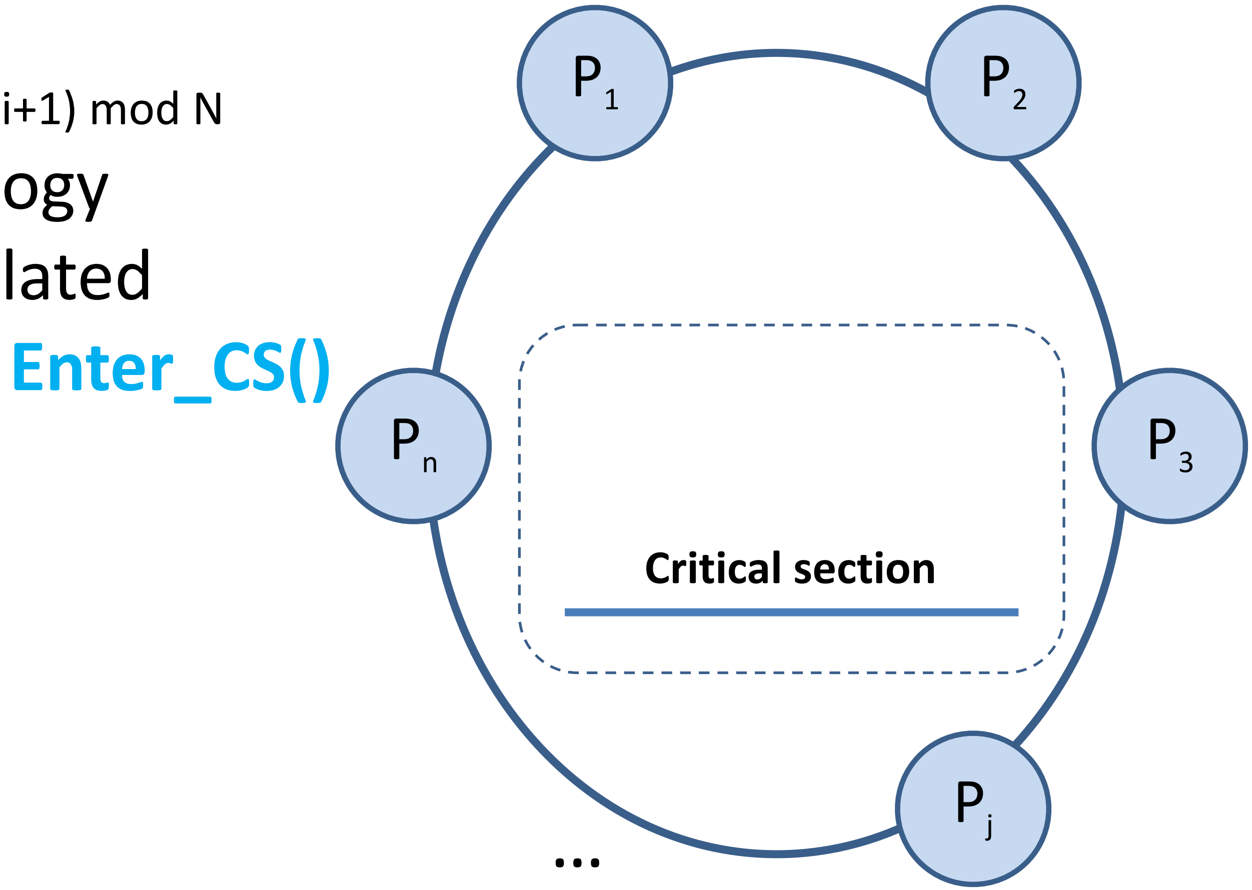
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



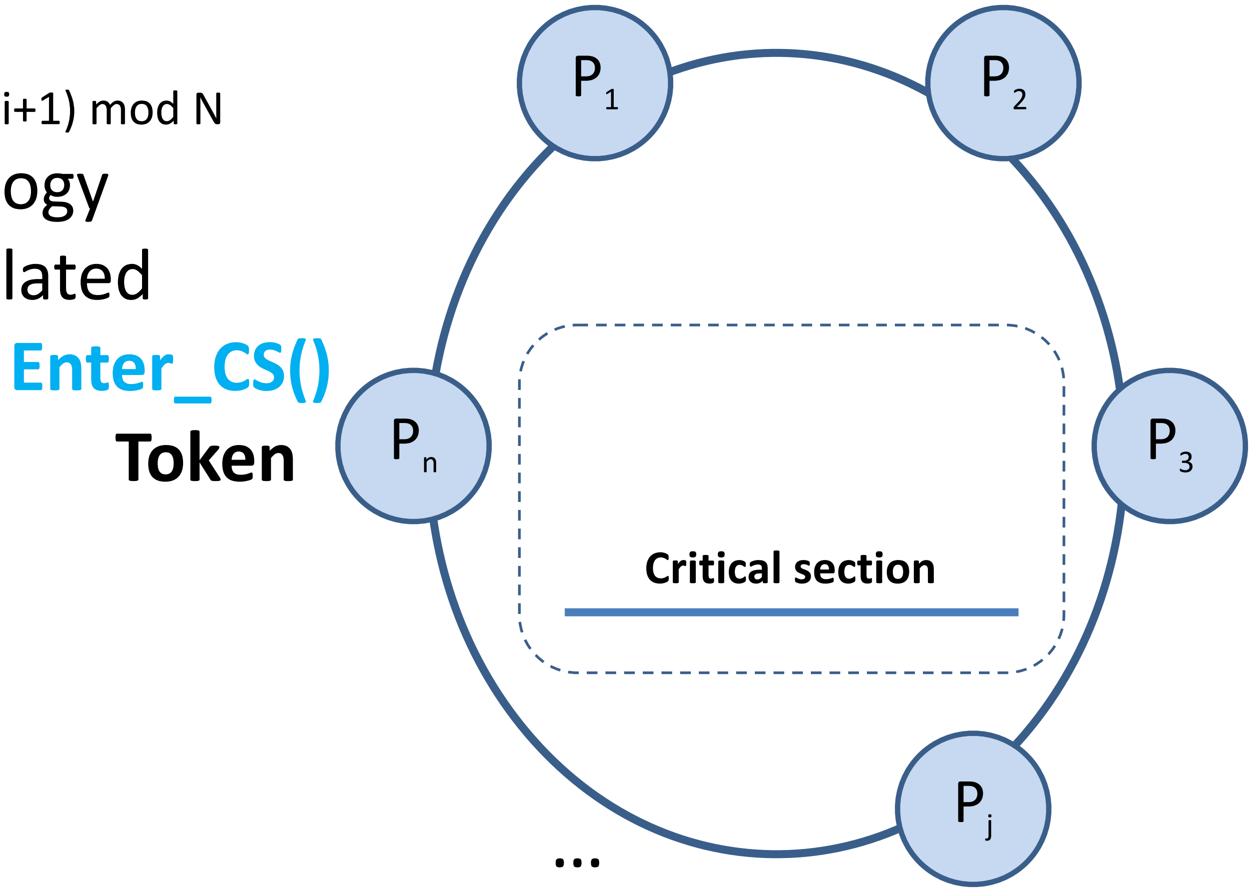
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



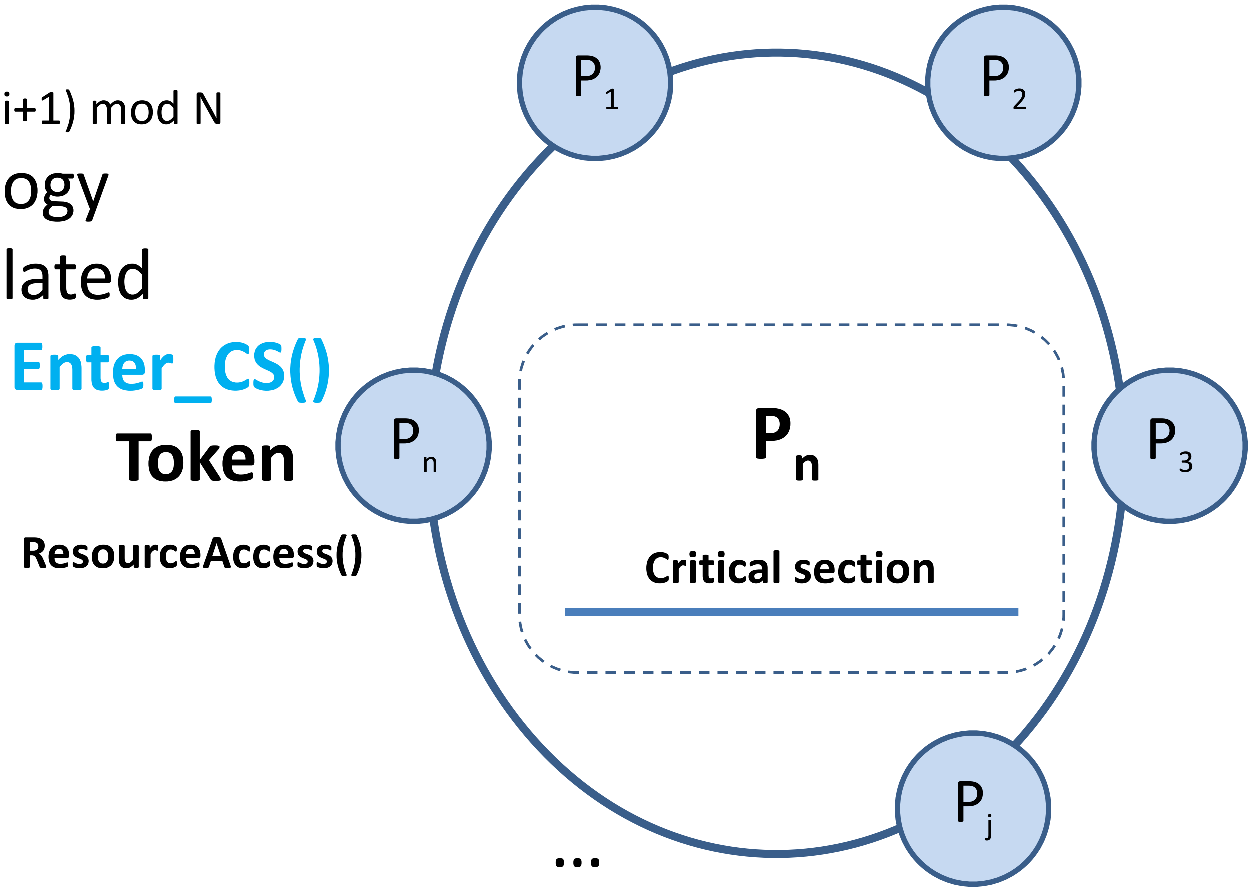
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



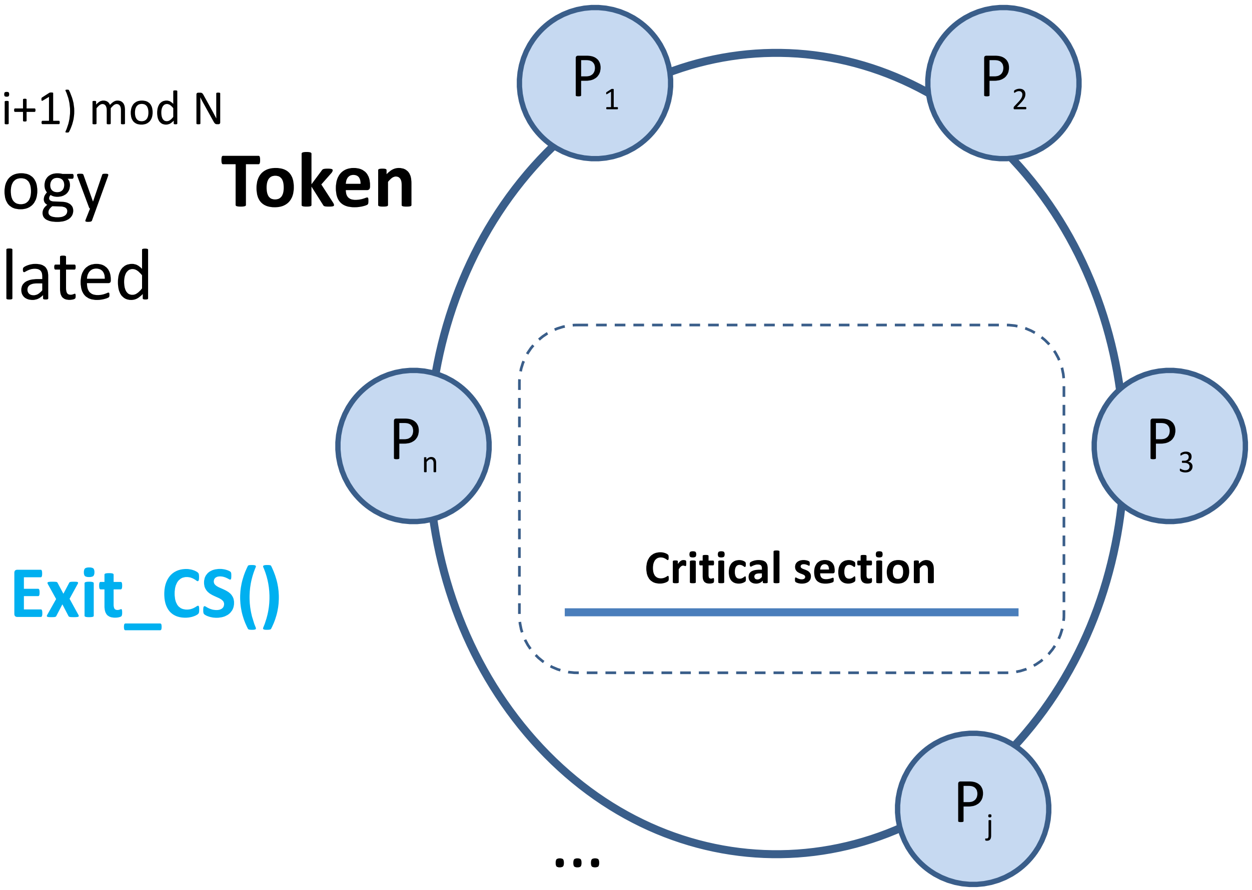
Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



Ring-based algorithm

- Logical ring of nodes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



Ring-based algorithm analysis

- **Safe**: Node enters CS only if it holds token
- **Live**: Since finite work is done by each node (can't re-enter), token eventually gets to each node
- **Fair**: Ordering is based on ring topology, no starvation (pass token between accesses)
- **Performance**
  - **Constantly consumes network bandwidth**, even when no node seeks entry, except when inside CS
  - **Synchronization delay**: Between 1 and *N* messages
  - **Client delay**: 0 to *N* messages for entry; 0 for exit

Potential problems with ring-based algorithm
(Due to our assumption, not all apply here.)

- Node crash
- Lost token
- Duplicate token
- Timeouts on token passing

Self-study questions

- Does solution meet the ordering requirement? Why or why not?
- How could access to multiple resources be realized with the ring-based ME algorithm?
- How can node failures be mitigated? Develop strategies that could tolerate 1, 2, … node failures?
- How could lost or duplicated tokens be mitigated?


Operating Systems — R. Stockton Gaines, Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

CR Categories: 4.32, 5.29

Introduction

The Partial Ordering

DISTRIBUTED MUTUAL EXCLUSION

LAMPORT’S ALGORITHM FOR MUTUAL EXCLUSION, 1978

Lamport's algorithm

Pixabay license

- System of *n* nodes
- (ts_i, i) – logical clock timestamp of node $\mathbf{P_i}$ **(total order!)**
- Logical timestamps serve to order requests for CS
- Smaller timestamps have priority over larger ones
- Request queue maintained at each node ordered by timestamp (**a priority queue!**)
- Assume message delivered in FIFO order, i.e., messages do not race over communication channel

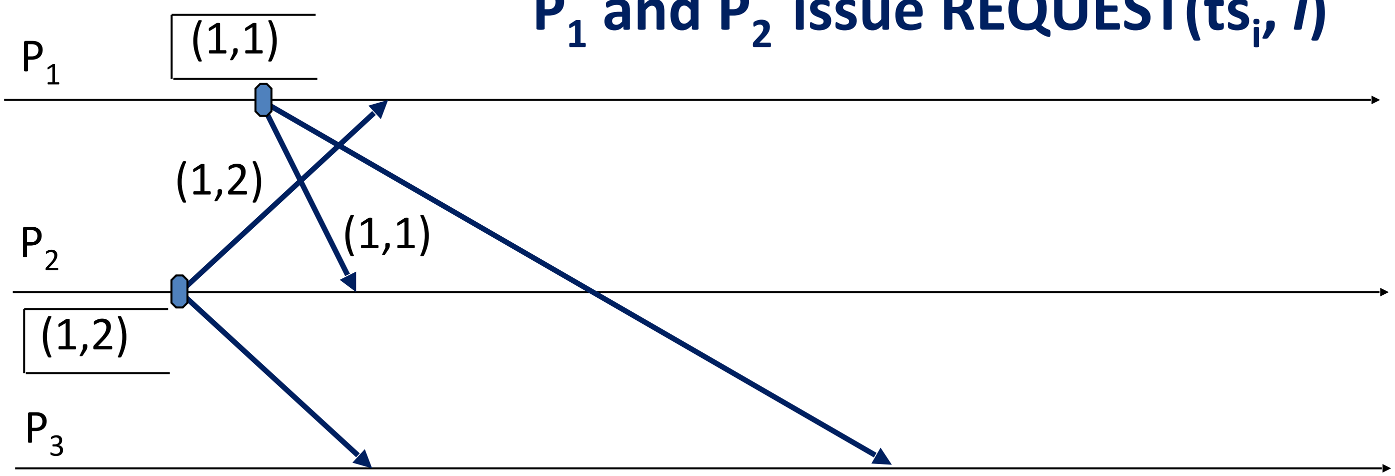
P_i Lamport's algorithm: Request

N.B.: (ts_i, *i*) denotes timestamp of request

P_i requesting CS

– Broadcast **REQUEST(ts_i, *i*) message** to ***all*** nodes
  - Place request in its ***request_queue$_i$***

P_1 and P_2 issue REQUEST(ts_i, *i*)

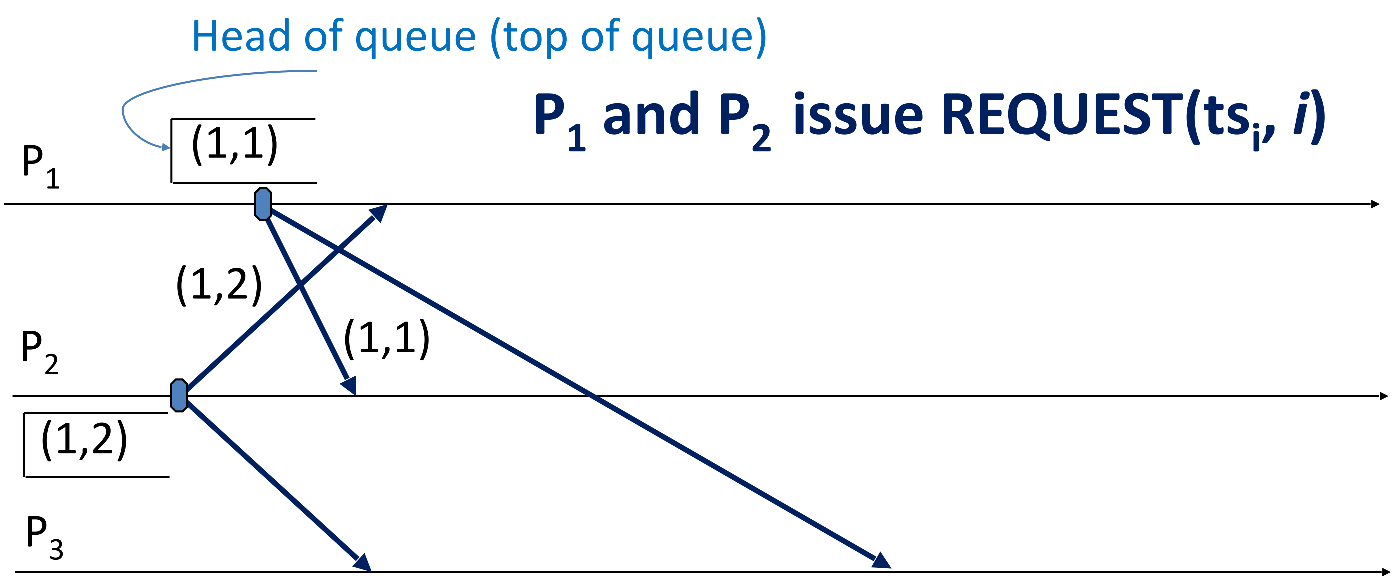


P_i Lamport's algorithm: Request

N.B.: (ts_i, *i*) denotes timestamp of request

P_i requesting CS

- Broadcast **REQUEST(ts_i, *i*) message** to ***all*** nodes
  - Place request in its ***request_queue$_i$***



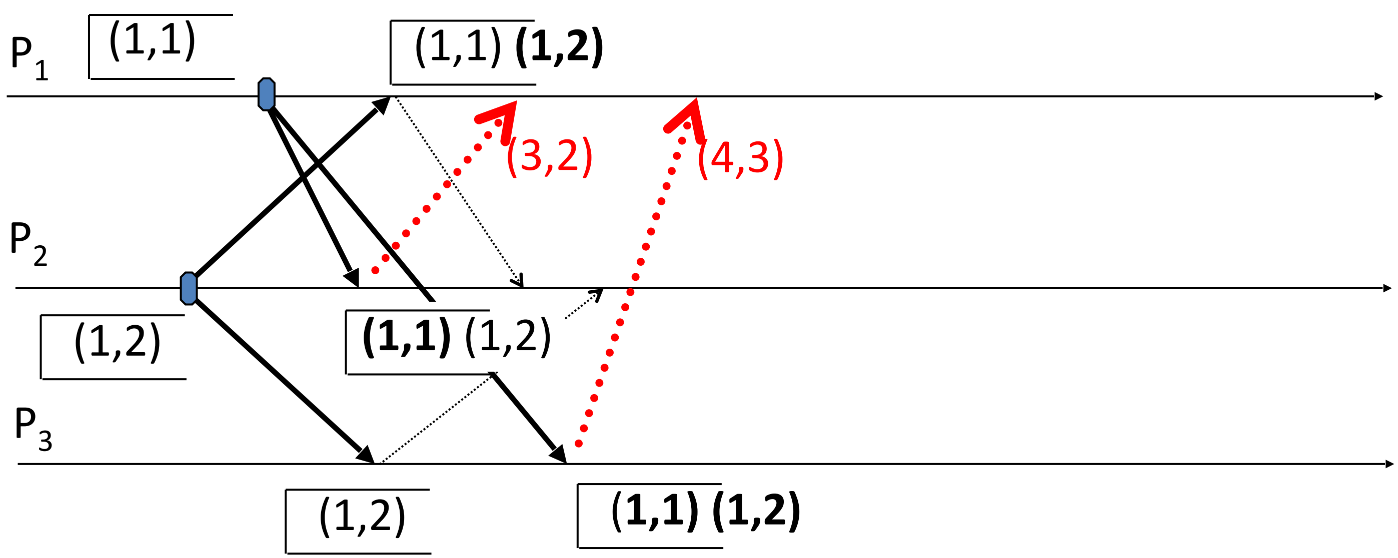
Lamport’s algorithm: Receive request

P_k

P_k receiving a request to enter CS

- When P_k receives a **REQUEST(ts_i, *i*) message** from P_i
  - It places P_i‘s request into its ***request_queue$_k$***
  - Sends a timestamped **REPLY message** to P_i

REPLY(ts_k, *k*) – only timestamped replies to P_i‘s request are shown below



P_i

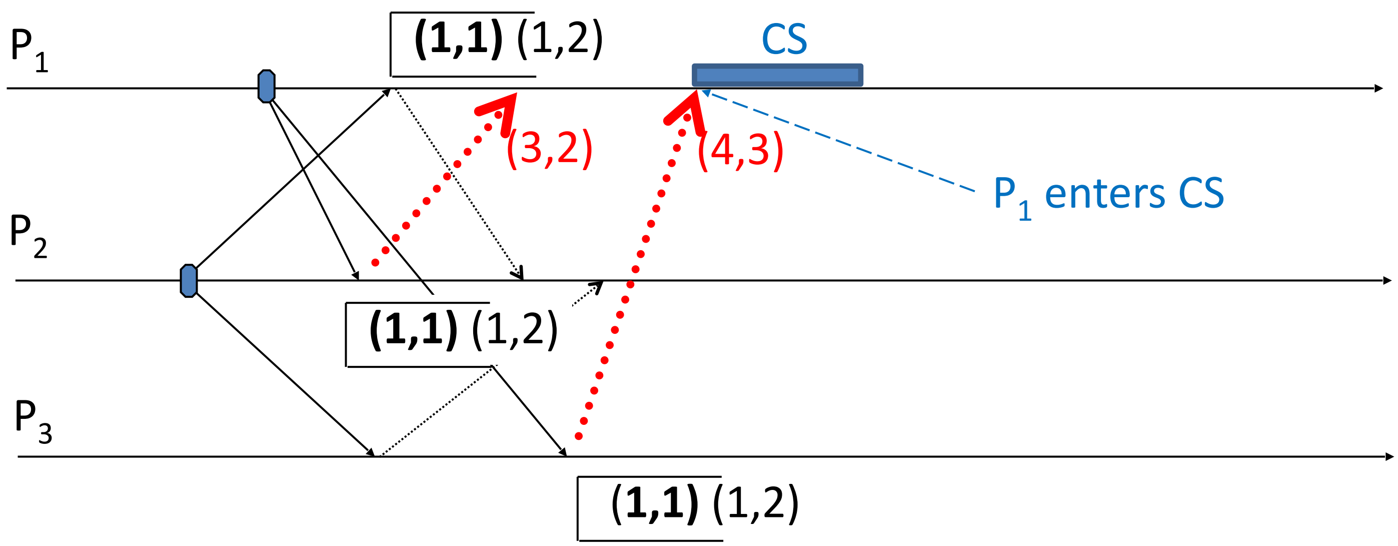
Lamport's algorithm

P_i requesting CS – **P_i enters CS when following conditions hold:**

1. **P_i** received a message with **timestamp larger than (ts_i, *i*) from every other node**
2. **P_i**'s request is at top of ***request_queue**$_i$*

Its original request timestamp

P_1 sent REQUEST(1, 1); now, it received REPLY(3,2) and REPLY(4,3), and its request (1, 1) is at the top of queue



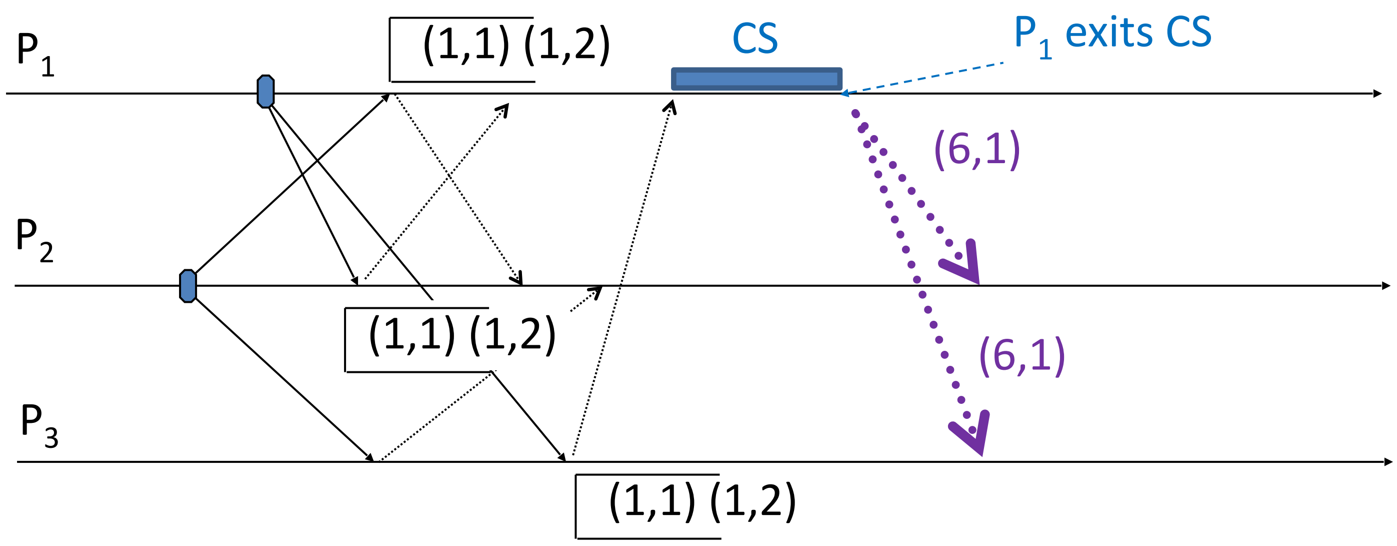
P_i

Lamport's algorithm

P_i releasing CS

- Removes its request from top of the request queue
- Broadcasts a timestamped **RELEASE** message to all nodes

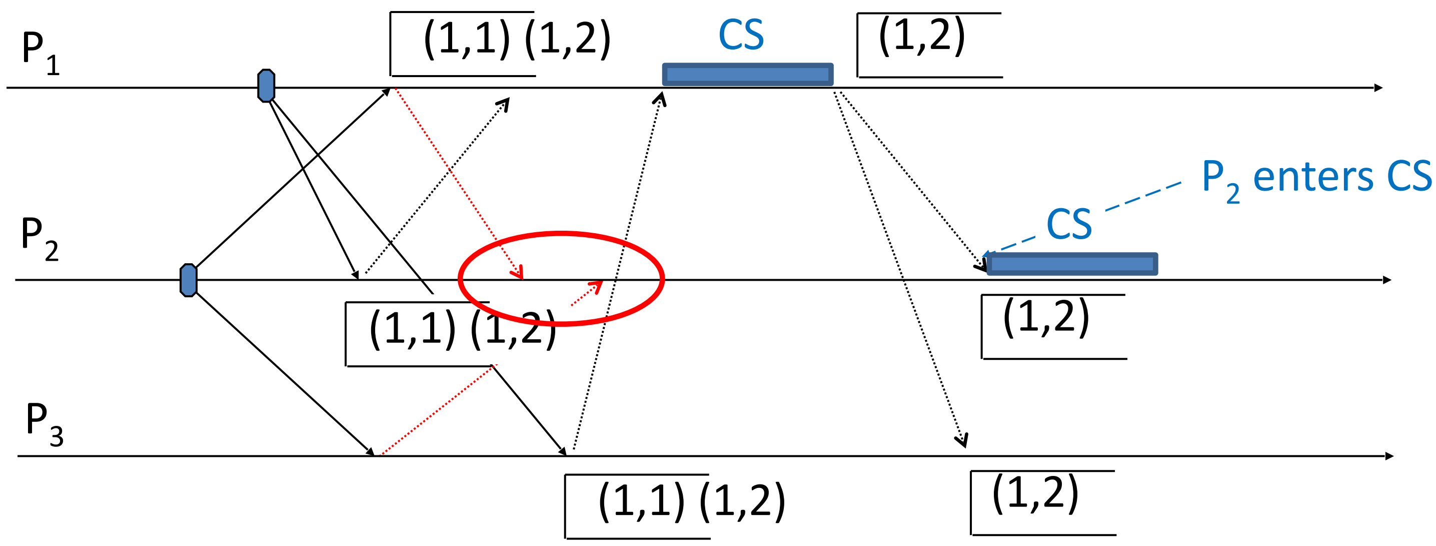
P_1 - RELEASE(ts_i, *i*)



P_k Lamport's algorithm

P_k receiving a release message

- When $\mathbf{P_k}$ receives a **RELEASE** message from $\mathbf{P_i}$, $\mathbf{P_k}$ removes $\mathbf{P_i}$'s request from its request queue



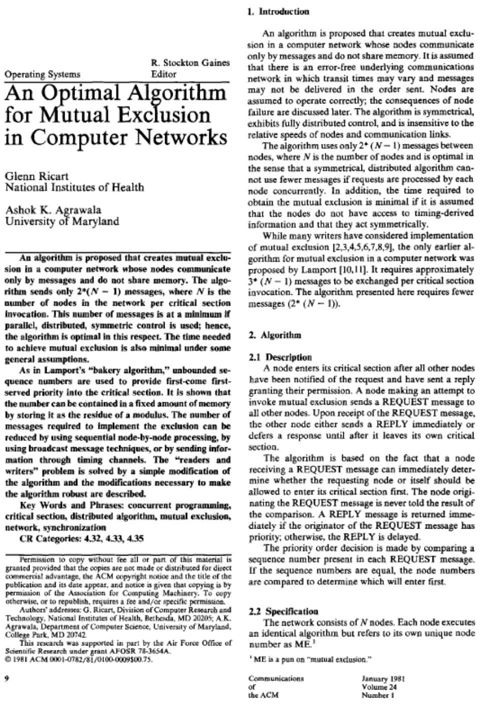
Summarizing observations

- 3(N-1) messages per CS request invocation
  - (N - 1) REQUEST
  - (N - 1) REPLY
  - (N - 1) RELEASE messages

- Not fault tolerant

Self-study questions

- Would node IDs as timestamps suffice, why or why not?
- Would Lamport clock values as timestamps suffice, why or why not?
- Is the algorithm order-preserving, why or why not?



Operating Systems

R. Stockton Gaines
Editor

An Optimal Algorithm for Mutual Exclusion in Computer Networks

Glenn Ricart
National Institutes of Health

Ashok K. Agrawala
University of Maryland

An algorithm is proposed that creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. The algorithm sends only 2*(N − 1) messages, where N is the number of nodes in the network per critical section invocation. This number of messages is at a minimum if parallel, distributed, symmetric control is used; hence, the algorithm is optimal in this respect. The time needed to achieve mutual exclusion is also minimal under some general assumptions.

As in Lamport's "bakery algorithm," unbounded sequence numbers are used to provide first-come first-served priority into the critical section. It is shown that the number can be contained in a fixed amount of memory by storing it as the residue of a modulus. The number of messages required to implement the exclusion can be reduced by using sequential node-by-node processing, by using broadcast message techniques, or by sending information through timing channels. The "readers and writers" problem is solved by a simple modification of the algorithm and the modifications necessary to make the algorithm robust are described.

Key Words and Phrases: concurrent programming, critical section, distributed algorithm, mutual exclusion, network, synchronization

CR Categories: 4.32, 4.33, 4.35

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Authors' addresses: G. Ricart, Division of Computer Research and Technology, National Institutes of Health, Bethesda, MD 20205; A.K. Agrawala, Department of Computer Science, University of Maryland, College Park, MD 20742.

This research was supported in part by the Air Force Office of Scientific Research under grant AFOSR 78-3654A.

© 1981 ACM 0001-0782/81/0100-0009$00.75.

9

1. Introduction

An algorithm is proposed that creates mutual exclusion in a computer network whose nodes communicate only by messages and do not share memory. It is assumed that there is an error-free underlying communications network in which transit times may vary and messages may not be delivered in the order sent. Nodes are assumed to operate correctly; the consequences of node failure are discussed later. The algorithm is symmetrical, exhibits fully distributed control, and is insensitive to the relative speeds of nodes and communication links.

The algorithm uses only 2* (N − 1) messages between nodes, where N is the number of nodes and is optimal in the sense that a symmetrical, distributed algorithm cannot use fewer messages if requests are processed by each node concurrently. In addition, the time required to obtain the mutual exclusion is minimal if it is assumed that the nodes do not have access to timing-derived information and that they act symmetrically.

While many writers have considered implementation of mutual exclusion [2,3,4,5,6,7,8,9], the only earlier algorithm for mutual exclusion in a computer network was proposed by Lamport [10,11]. It requires approximately 3* (N − 1) messages to be exchanged per critical section invocation. The algorithm presented here requires fewer messages (2* (N − 1)).

2. Algorithm

2.1 Description

A node enters its critical section after all other nodes have been notified of the request and have sent a reply granting their permission. A node making an attempt to invoke mutual exclusion sends a REQUEST message to all other nodes. Upon receipt of the REQUEST message, the other node either sends a REPLY immediately or defers a response until after it leaves its own critical section.

The algorithm is based on the fact that a node receiving a REQUEST message can immediately determine whether the requesting node or itself should be allowed to enter its critical section first. The node originating the REQUEST message is never told the result of the comparison. A REPLY message is returned immediately if the originator of the REQUEST message has priority; otherwise, the REPLY is delayed.

The priority order decision is made by comparing a sequence number present in each REQUEST message. If the sequence numbers are equal, the node numbers are compared to determine which will enter first.

2.2 Specification

The network consists of N nodes. Each node executes an identical algorithm but refers to its own unique node number as ME.[1]

[1] ME is a pun on "mutual exclusion."

Communications of the ACM — January 1981, Volume 24, Number 1

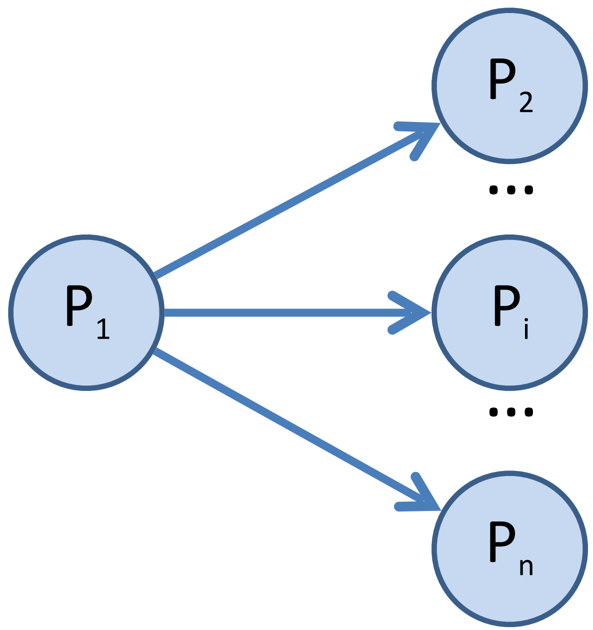
DISTRIBUTED MUTUAL EXCLUSION

RICART & AGRAWALA, 1981

Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among *n* node)

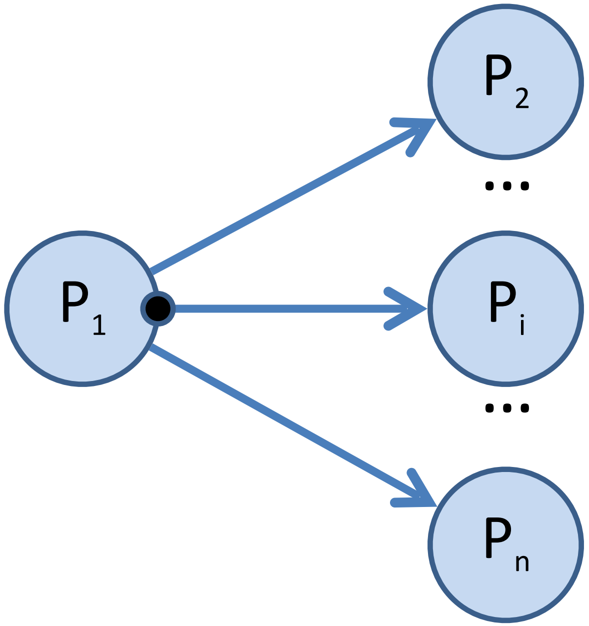
- **Basic idea**
  - Nodes wanting to enter CS, **broadcast a request to all other nodes**
  - Enter CS, once **all** nodes have **granted request**
- Use **Lamport timestamps** to order requests: <ts_i, *i*>, ts_i the timestamp, *i* the node identifier



Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among *n* node)

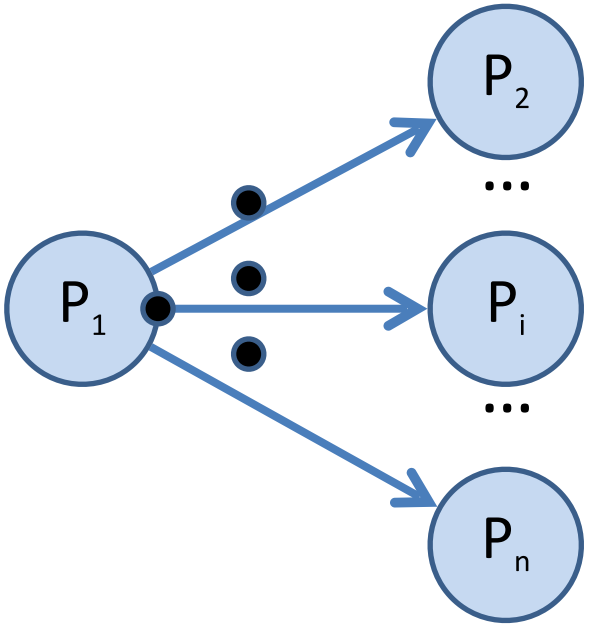
- **Basic idea**
  - Nodes wanting to enter CS, **broadcast a request to all other nodes**
  - Enter CS, once **all** nodes have **granted request**
- Use **Lamport timestamps** to order requests: <ts_i, i>, ts_i the timestamp, i the node identifier



Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among *n* node)

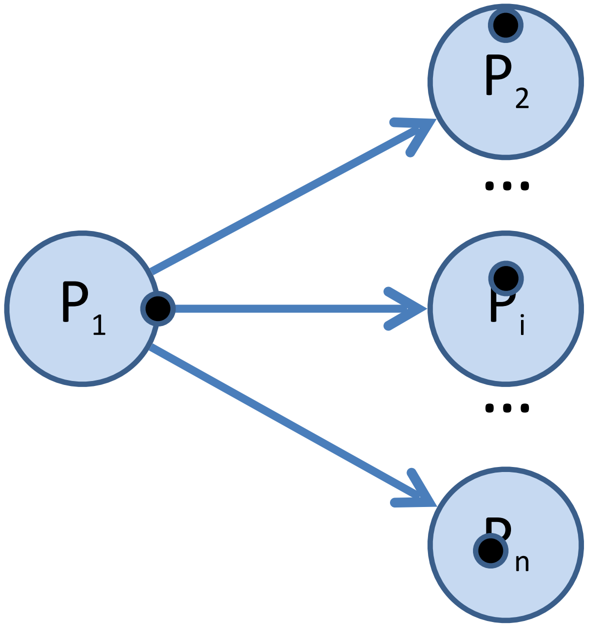
- **Basic idea**
  - Nodes wanting to enter CS, **broadcast a request to all other nodes**
  - Enter CS, once **all** nodes have **granted request**
- Use **Lamport timestamps** to order requests: <ts_i, i>, ts_i the timestamp, i the node identifier



Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among *n* node)

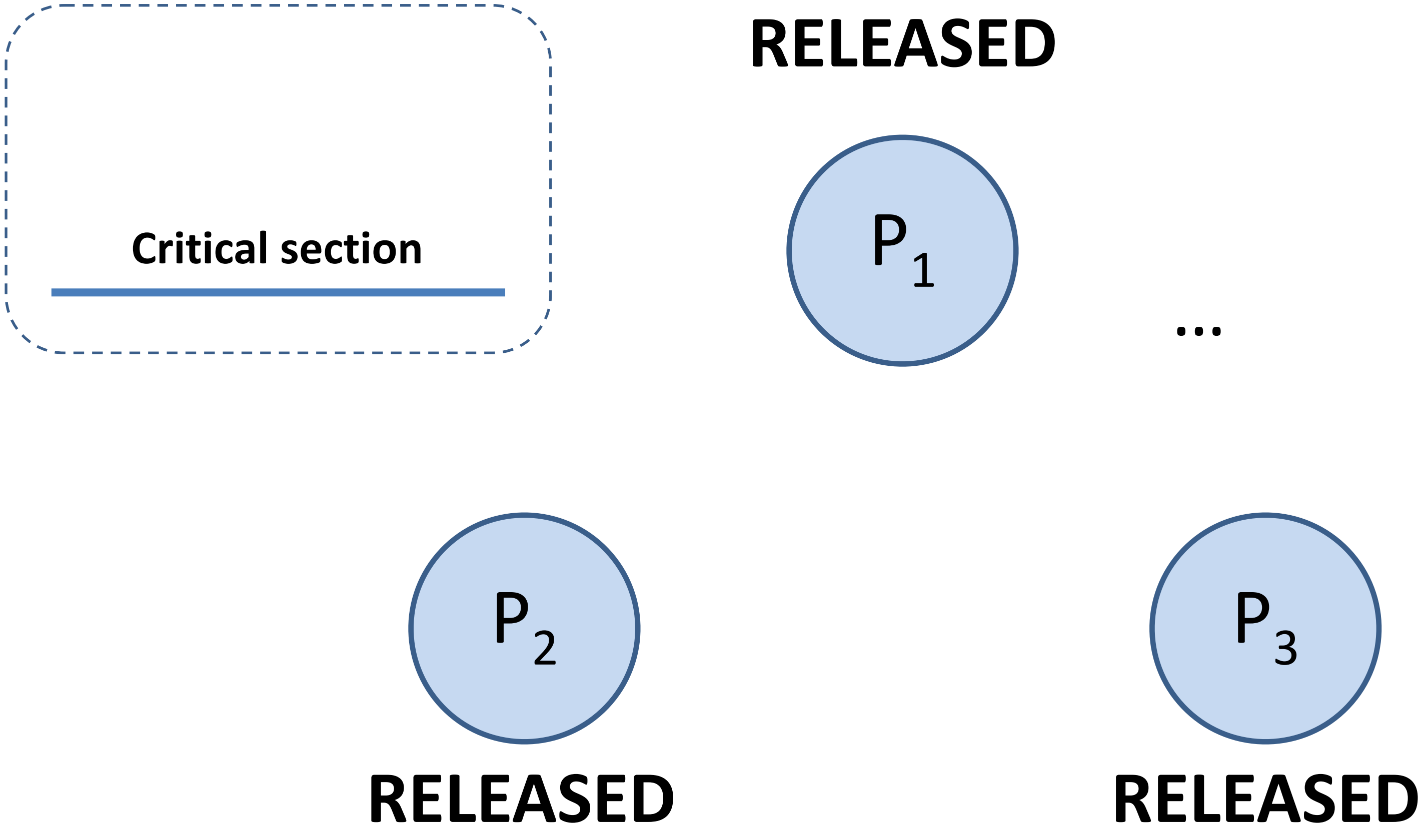
- **Basic idea**
  - Nodes wanting to enter CS, **broadcast a request to all other nodes**
  - Enter CS, once **all** nodes have **granted request**
- Use **Lamport timestamps** to order requests: <ts_i, *i*>, ts_i the timestamp, *i* the node identifier



Ricart & Agrawala: Distributed strategy

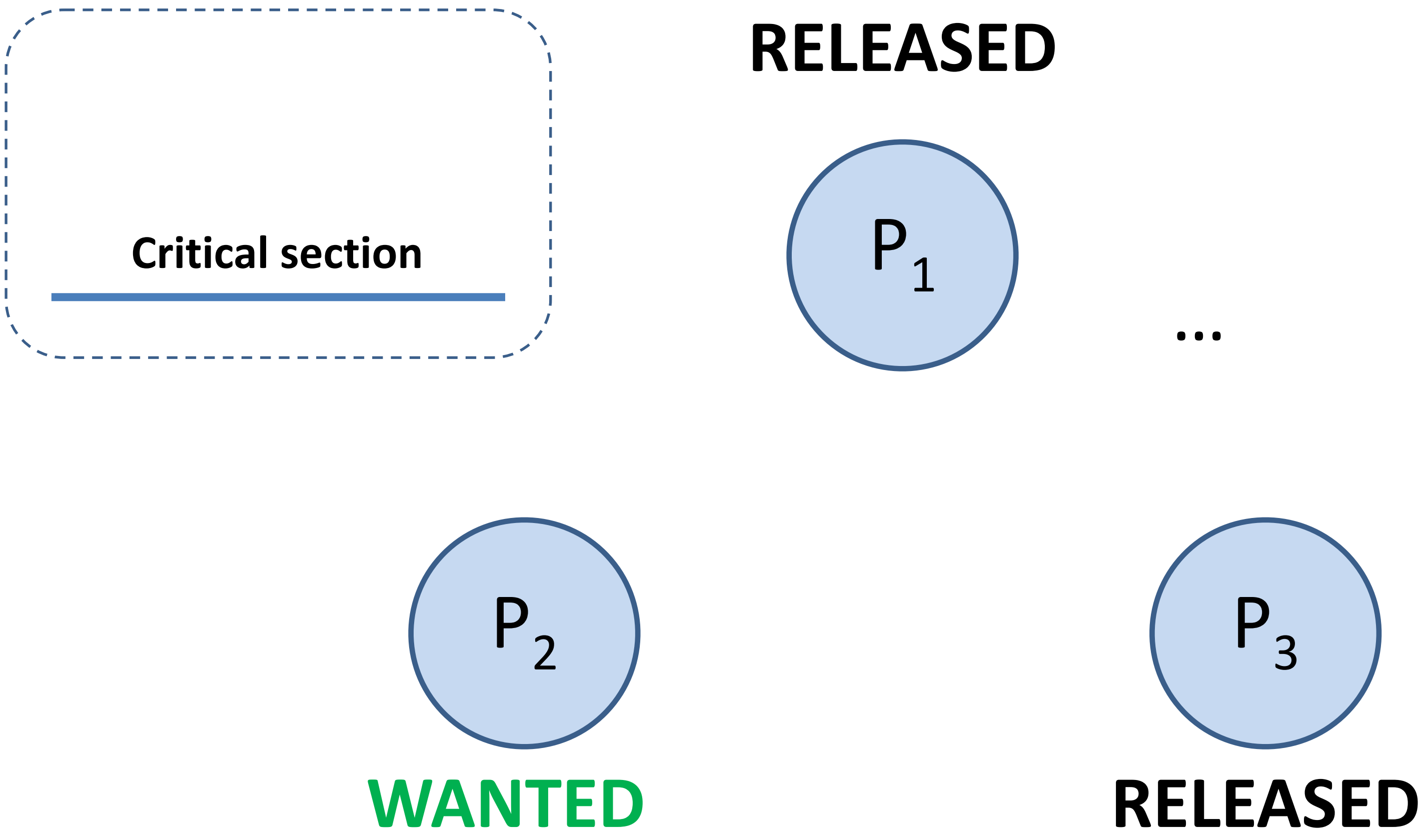
- Each node is in one of three states
  - **Released** - Outside the CS, e.g., after Exit_CS()
  - **Wanted** - Wanting to enter CS, in call Enter_CS()
  - **Held** - Inside CS, during RessourceAccess()
- If a node requests to enter CS and **all** other **nodes** are **in the *Released state***, **entry** is **granted** by each node
- If a node, $P_{i,}$ requests to enter CS and another node, $P_{k,}$ is inside the CS (***Held state***), then P_k will not reply, until it is finished with the CS

Initialization



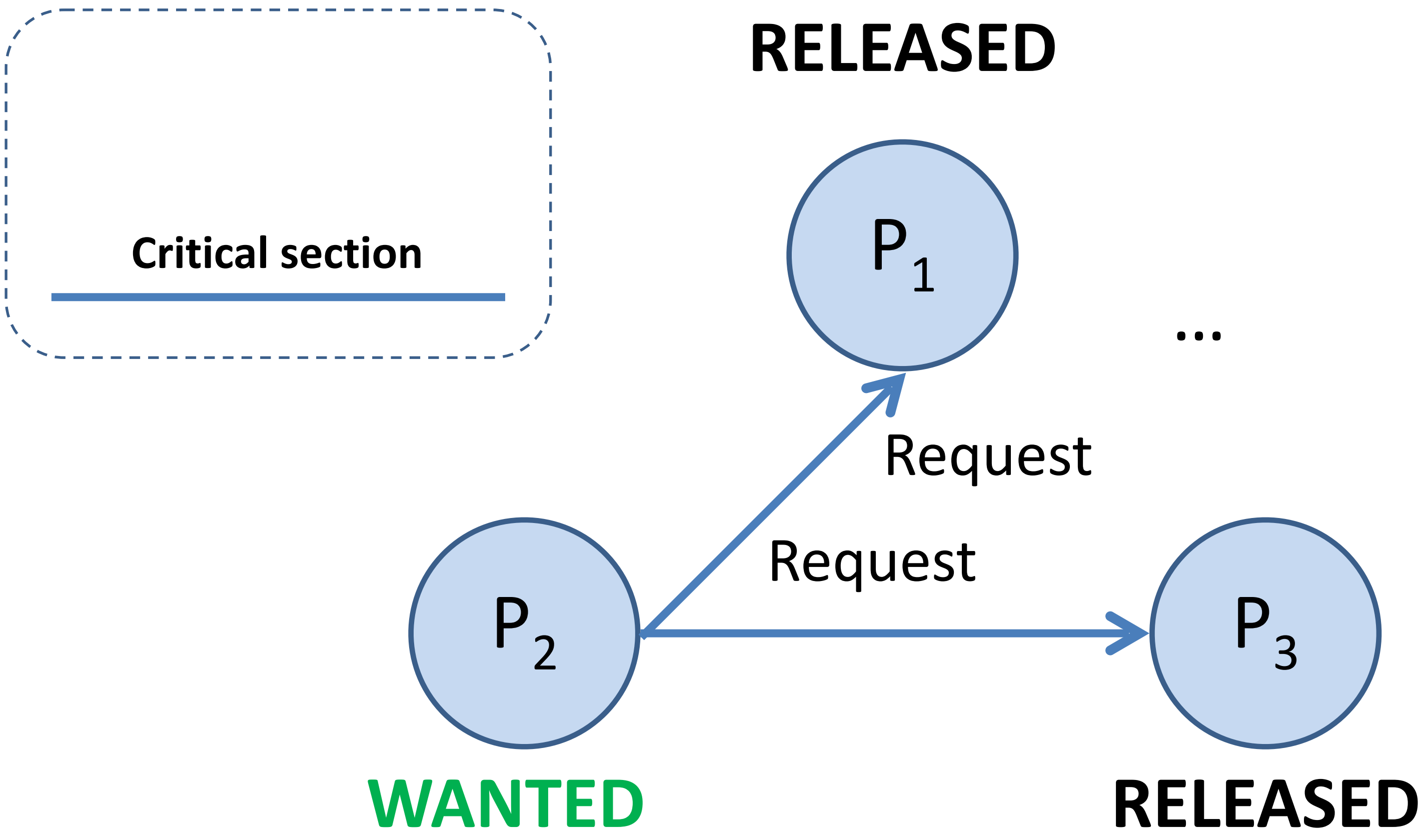
Requesting entry to CS

Request while *all Released*



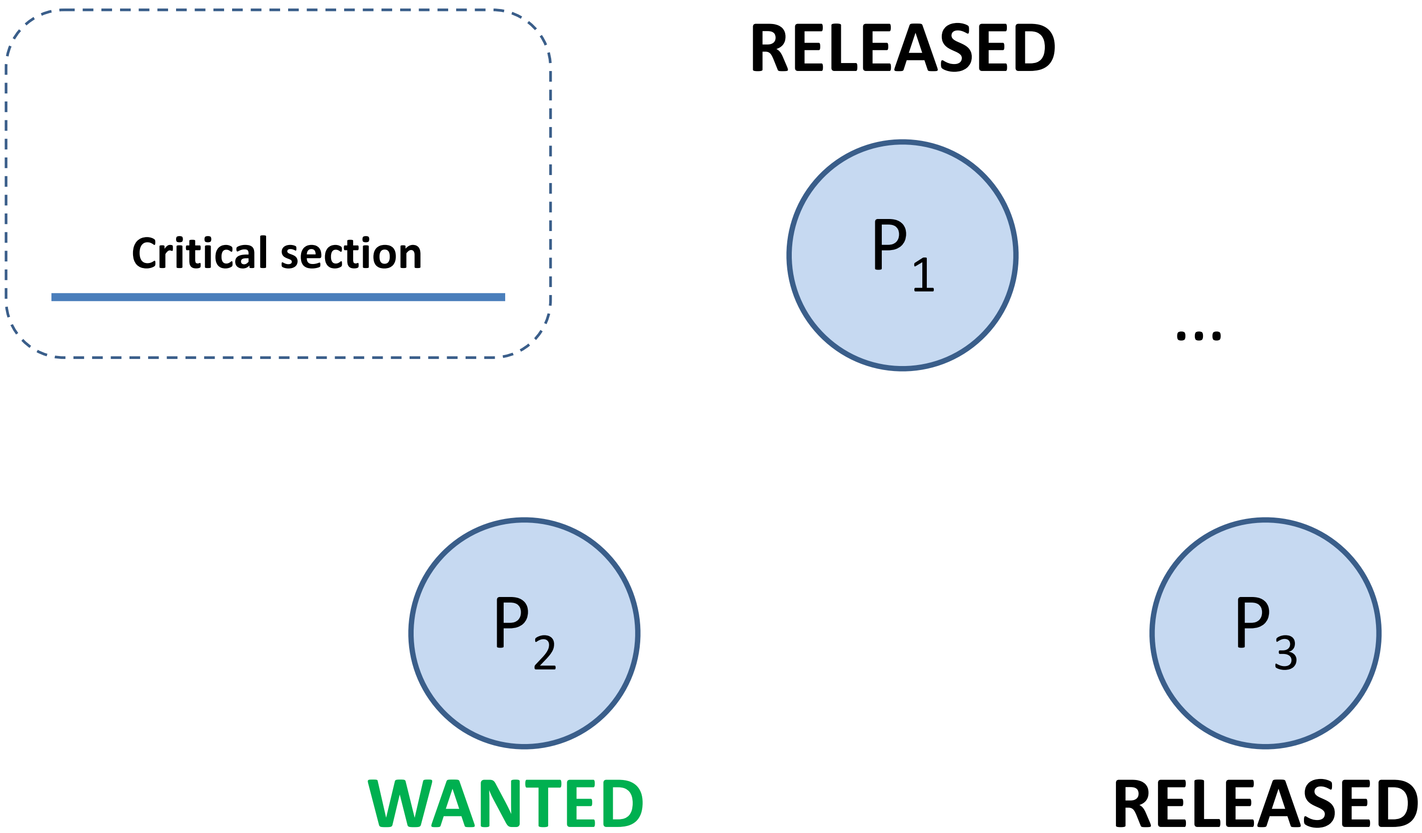
Requesting entry to CS

Request while *all Released*



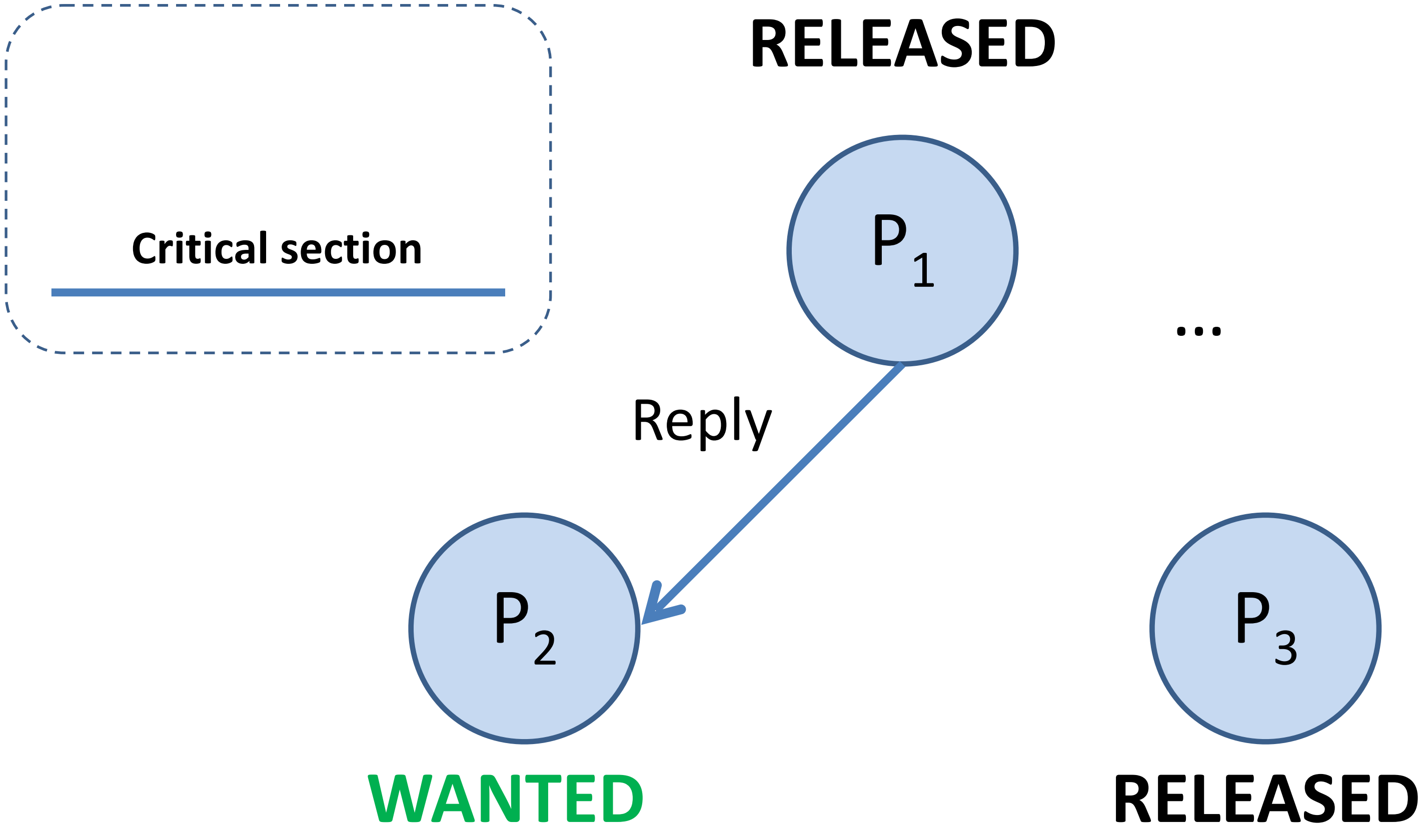
Requesting entry to CS

Request while *all Released*



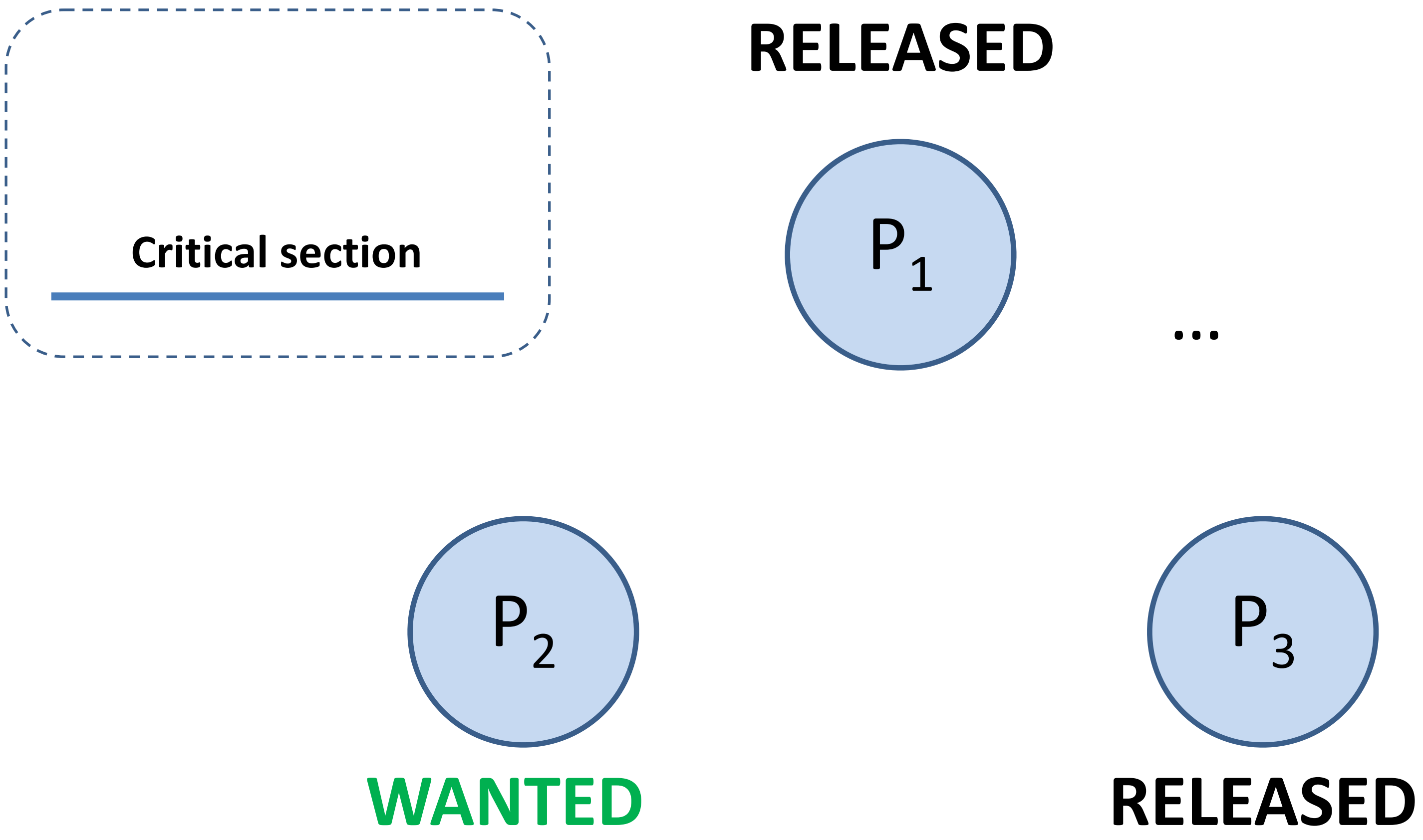
Requesting entry to CS

Request while *all Released*



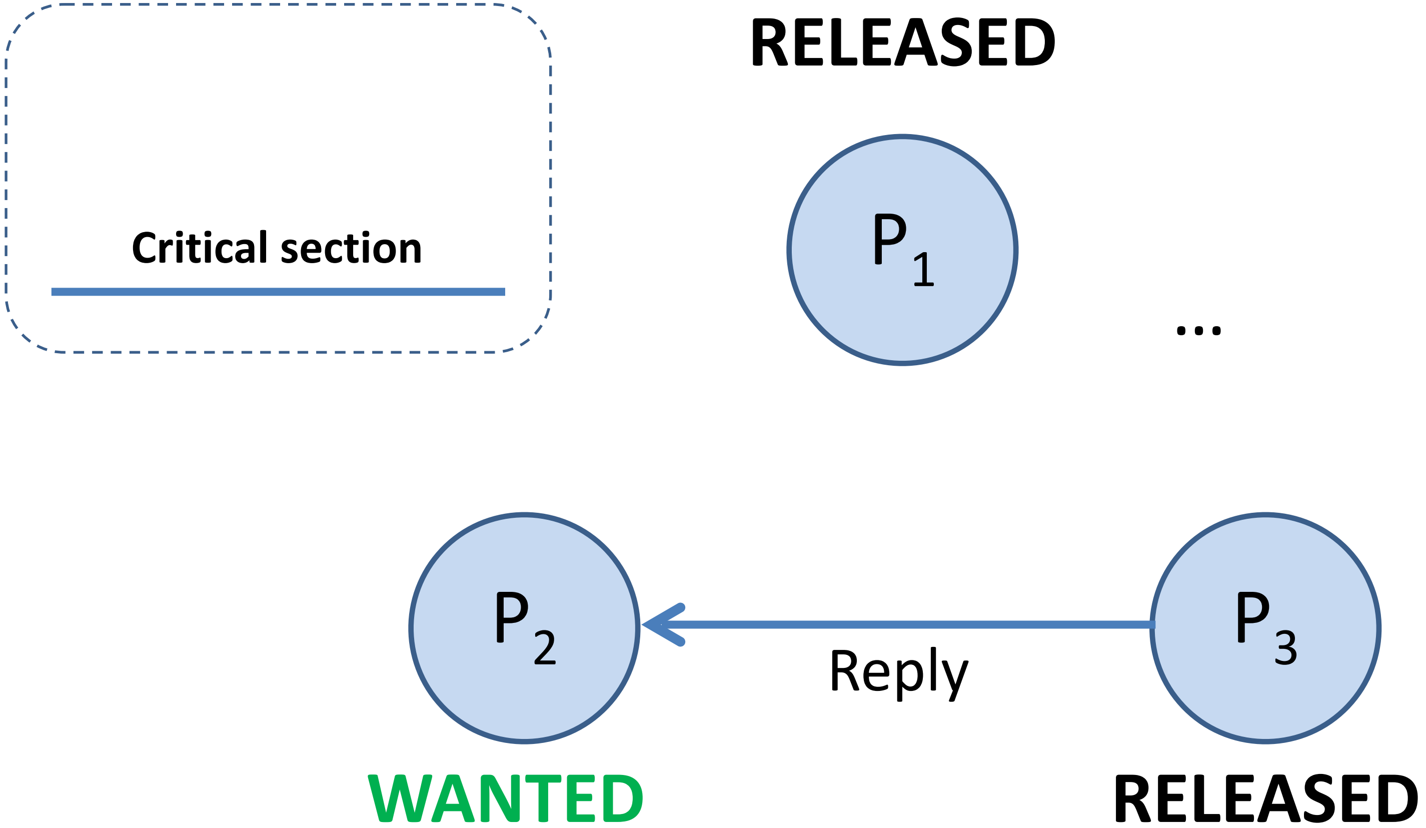
Requesting entry to CS

Request while *all Released*



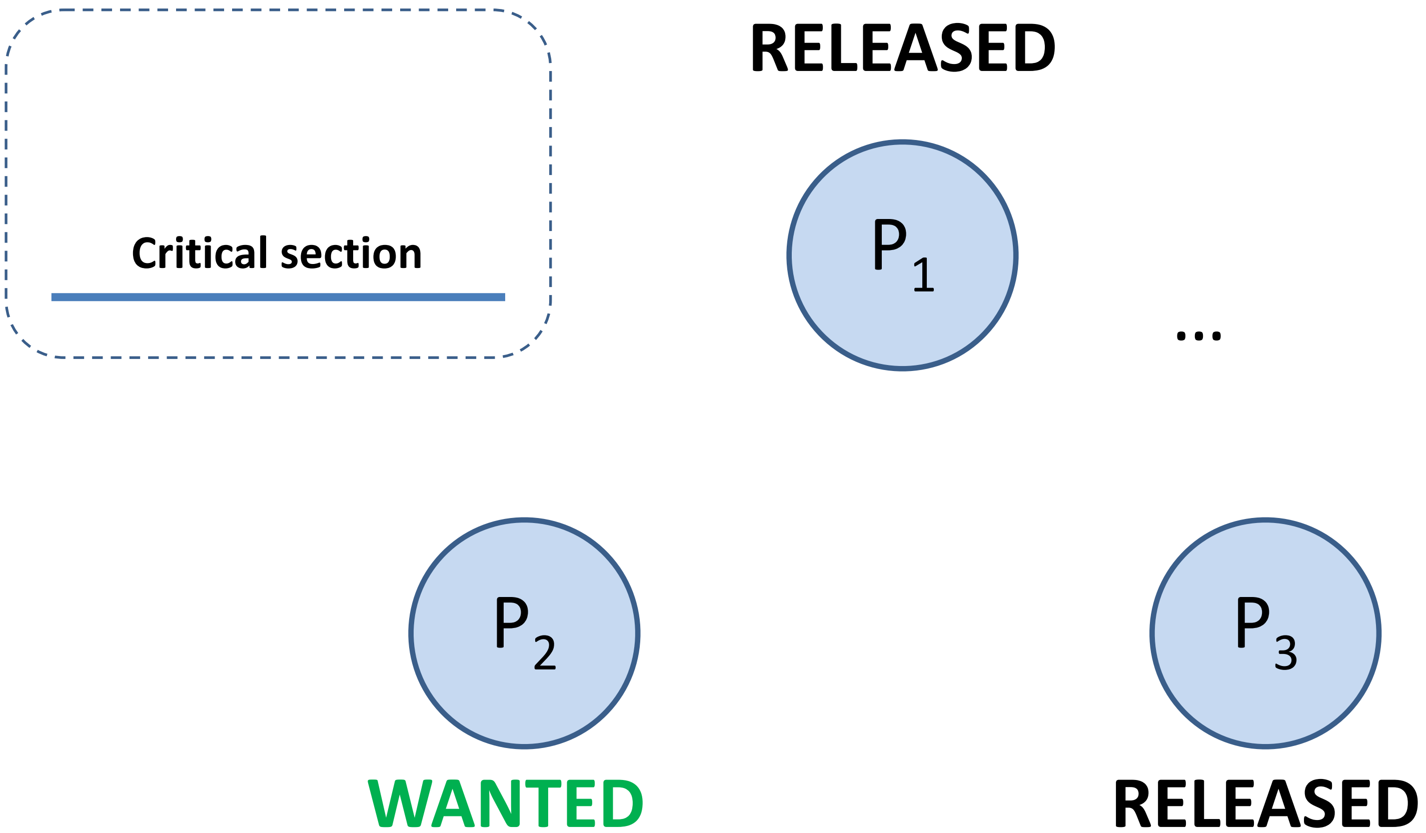
Requesting entry to CS

Request while *all Released*



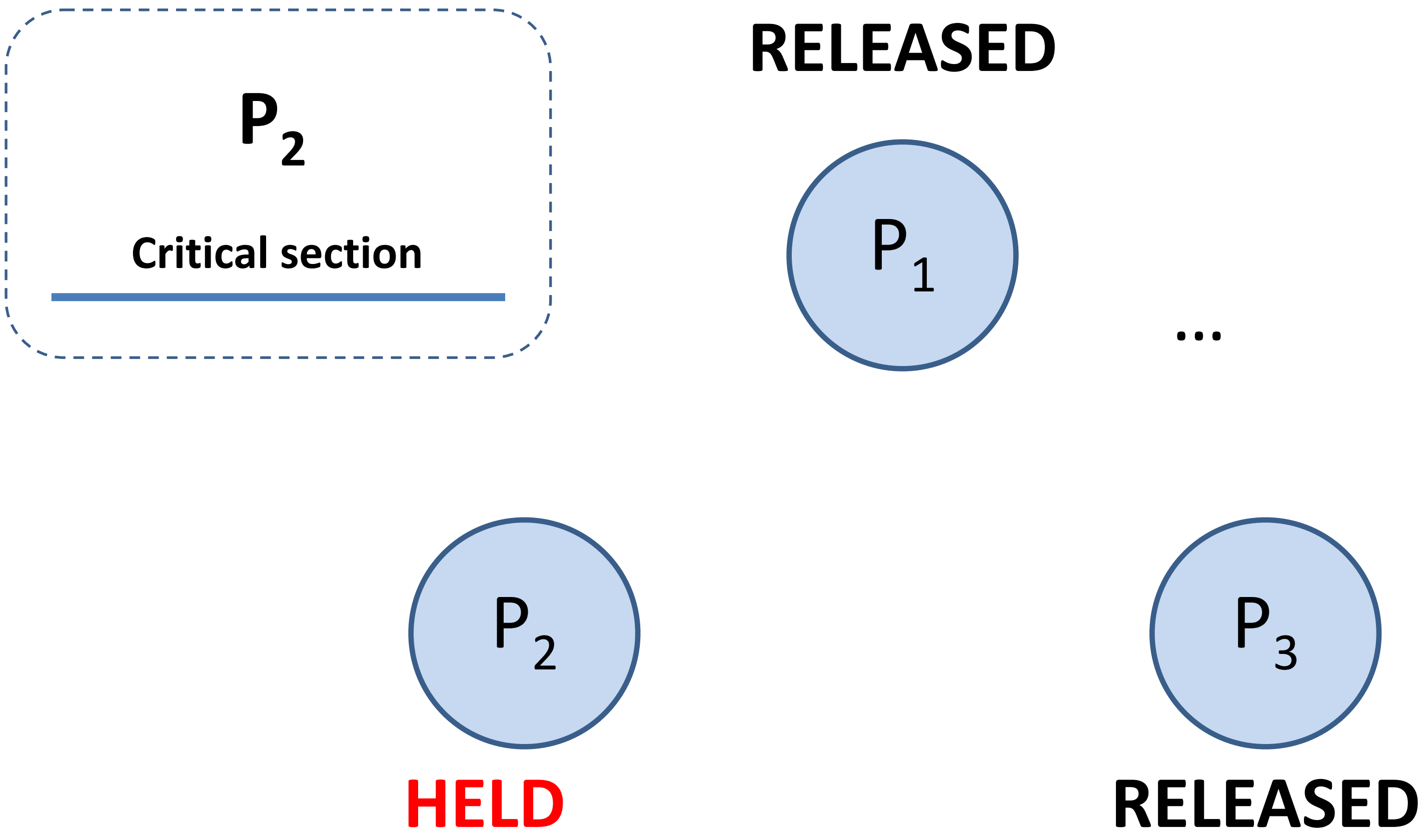
Requesting entry to CS

Request while *all Released*



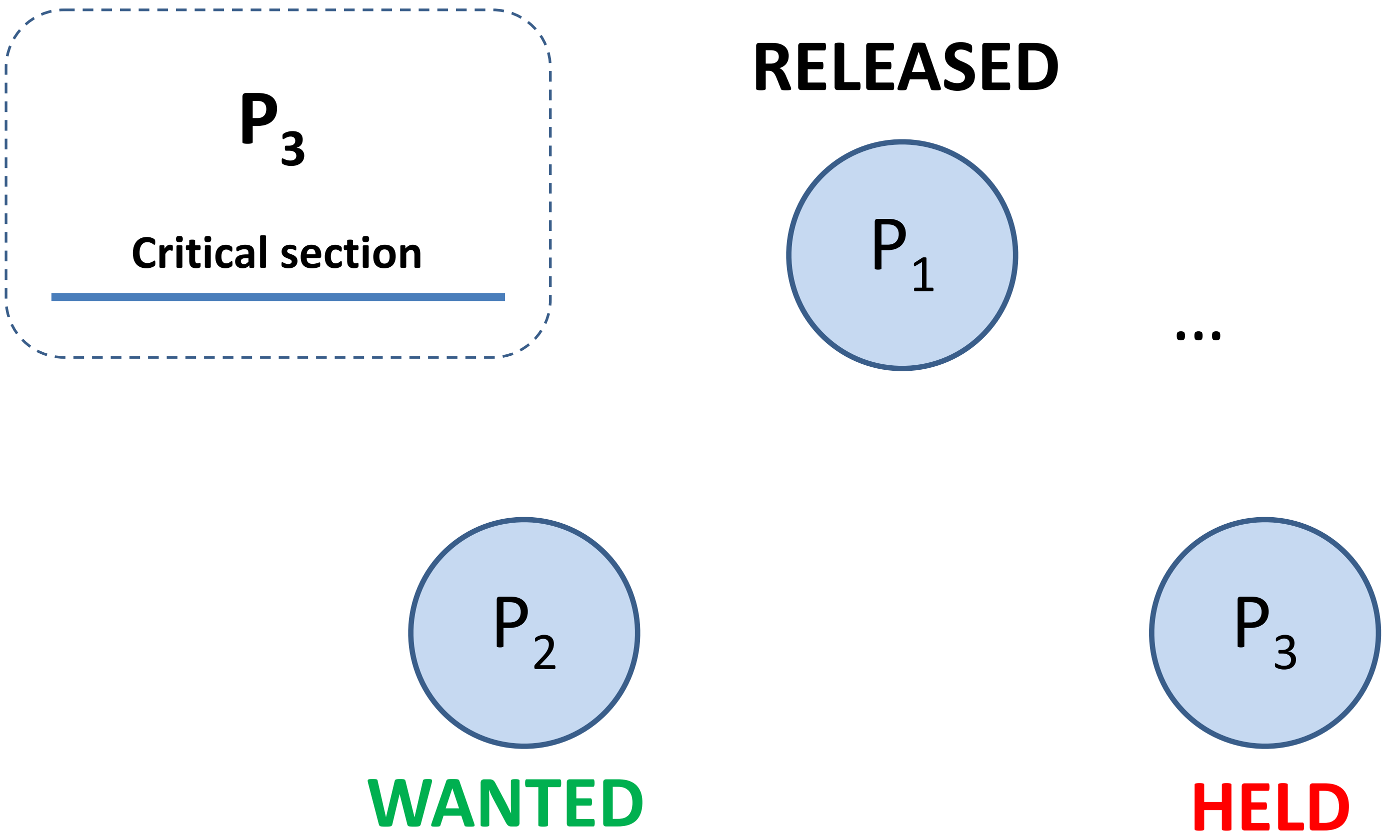
Requesting entry to CS

Request while *all Released*



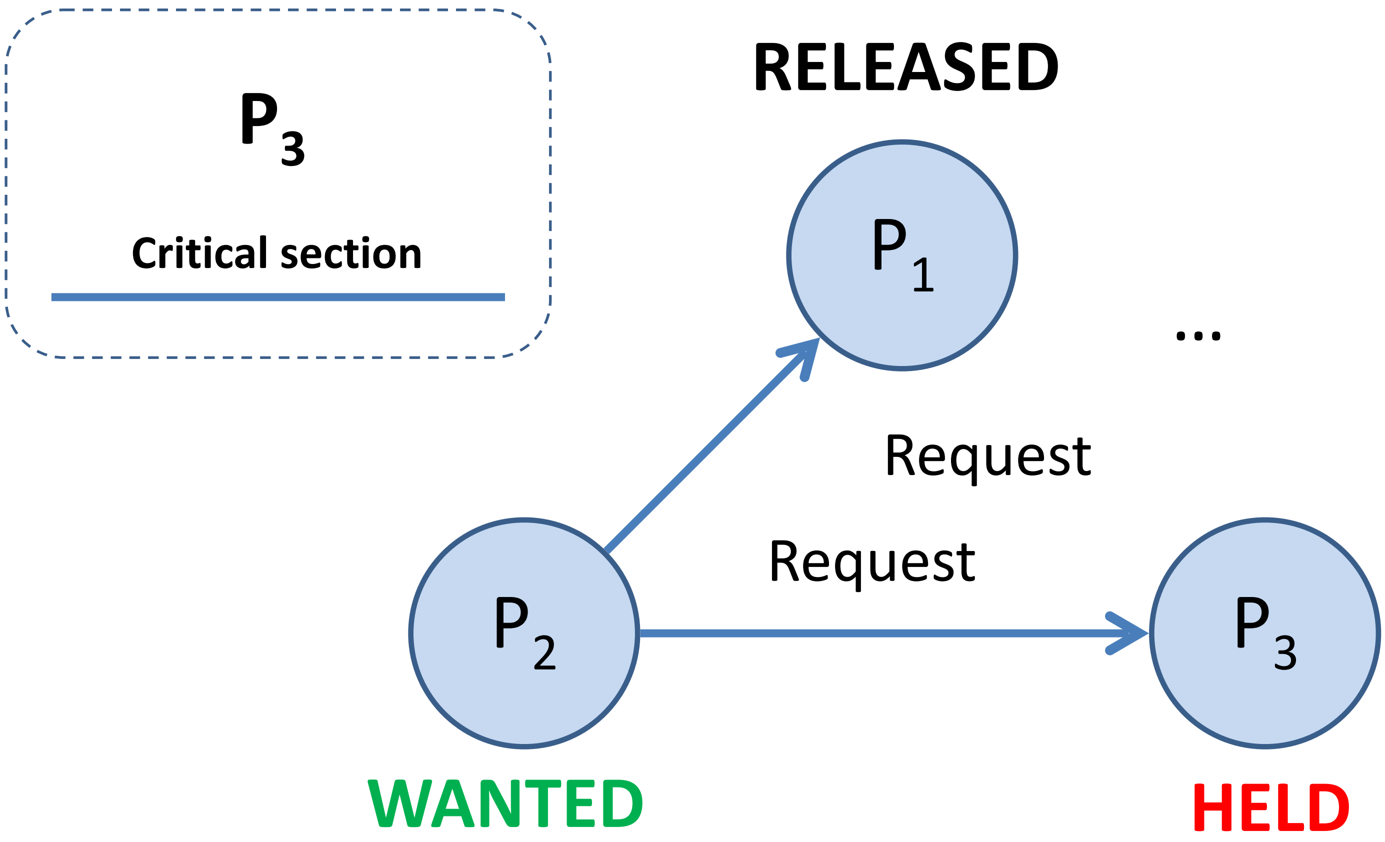
Requesting entry to CS

Request while Held



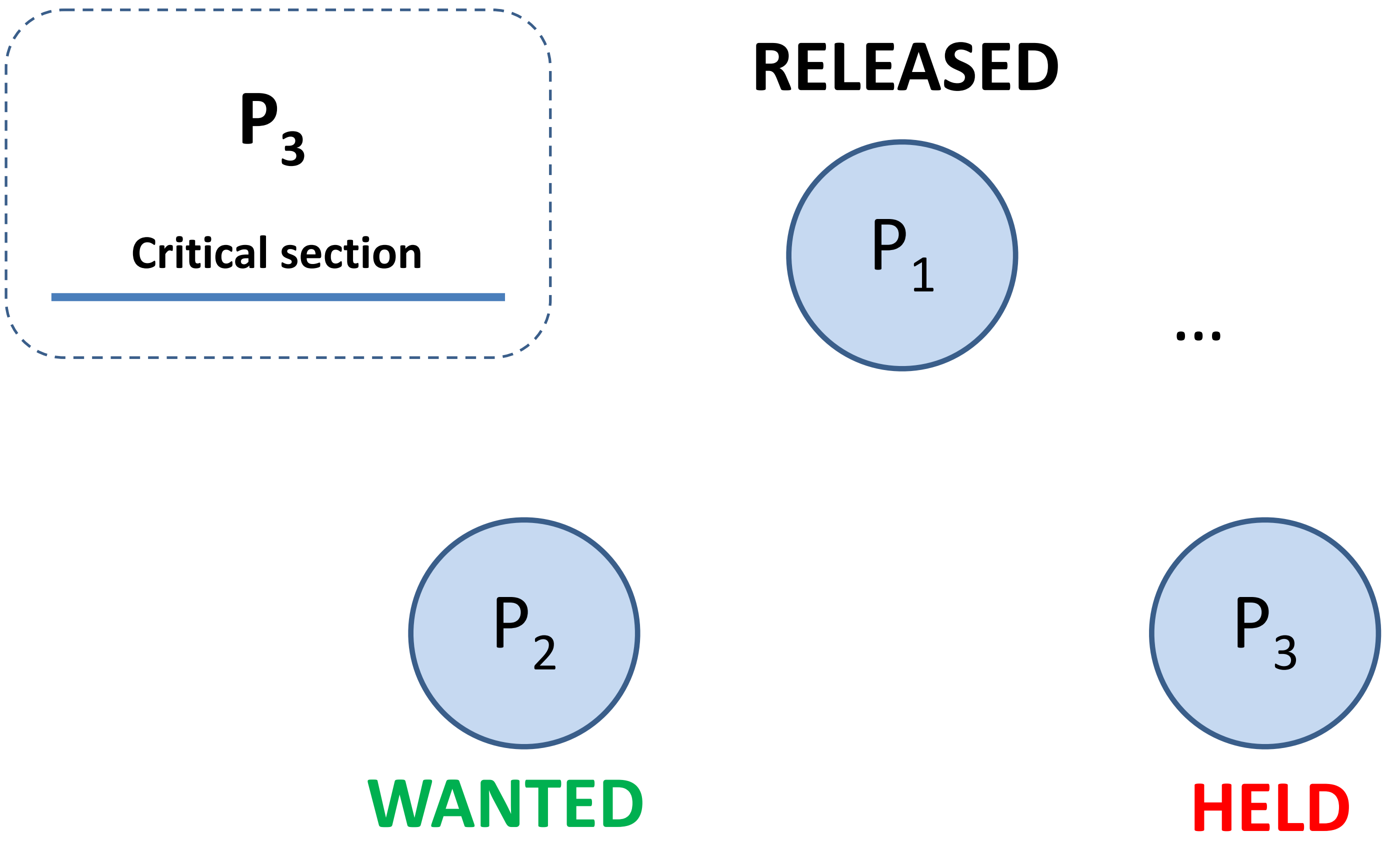
Requesting entry to CS

Request while Held



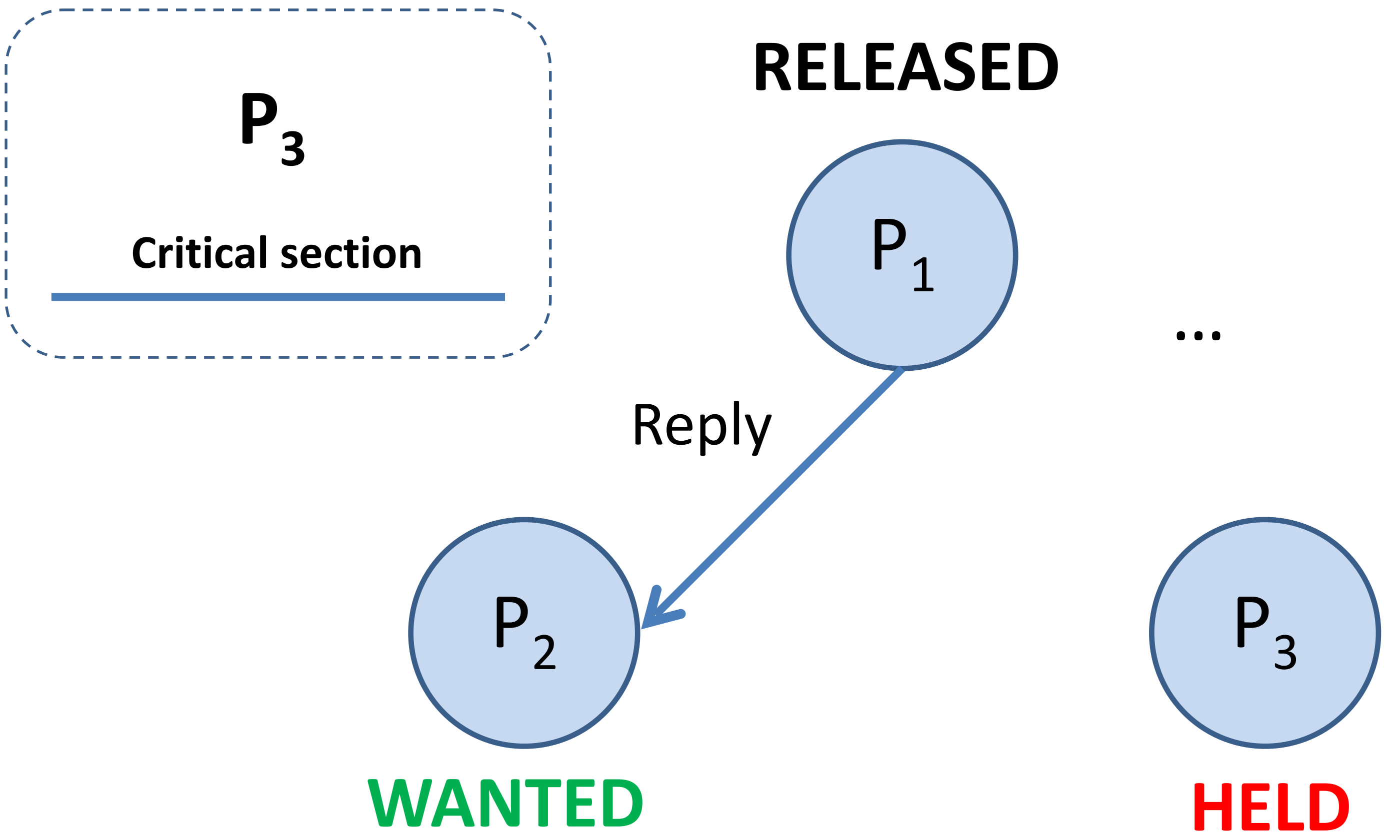
Requesting entry to CS

Request while Held



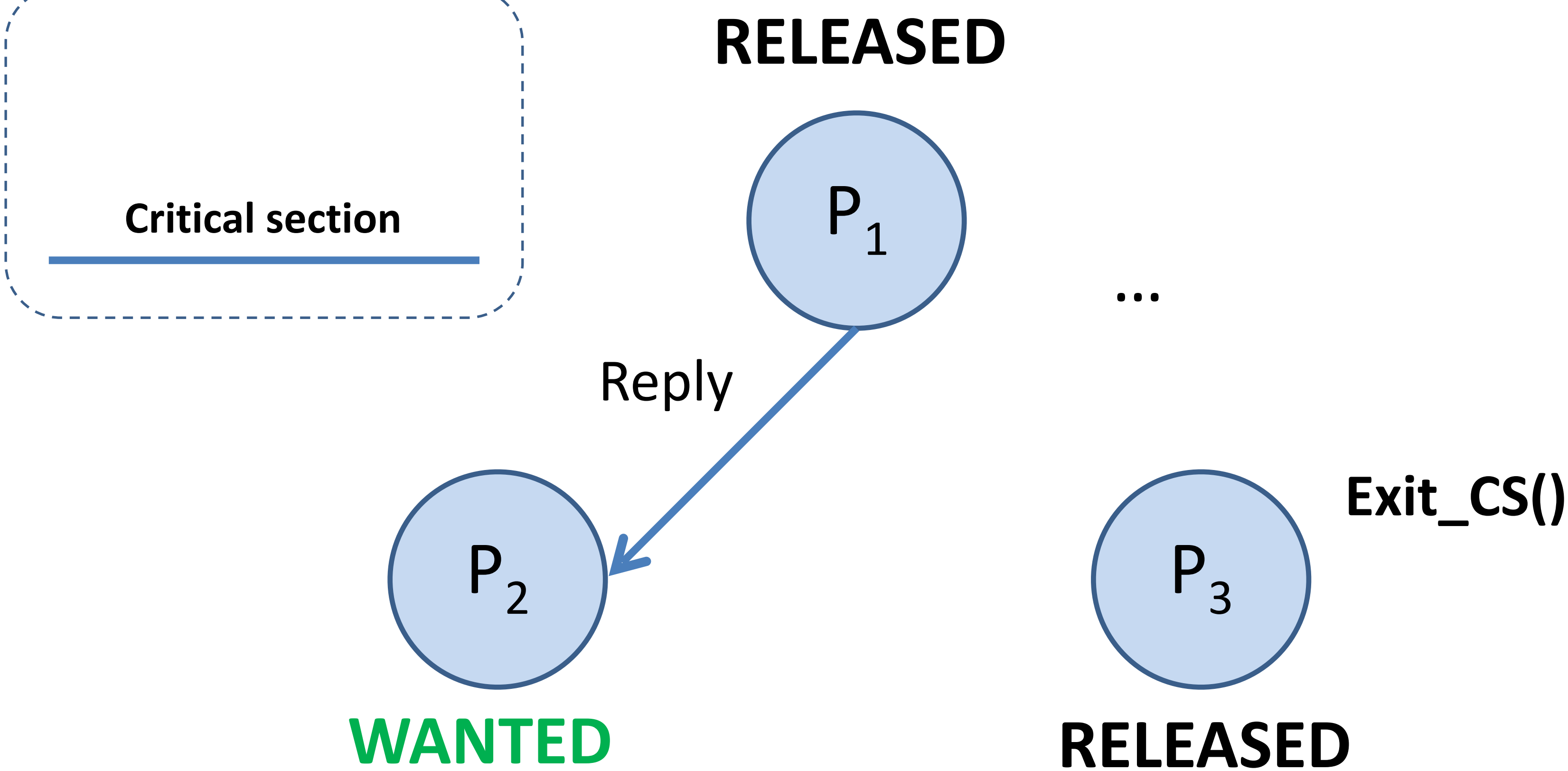
Requesting entry to CS

Request while Held



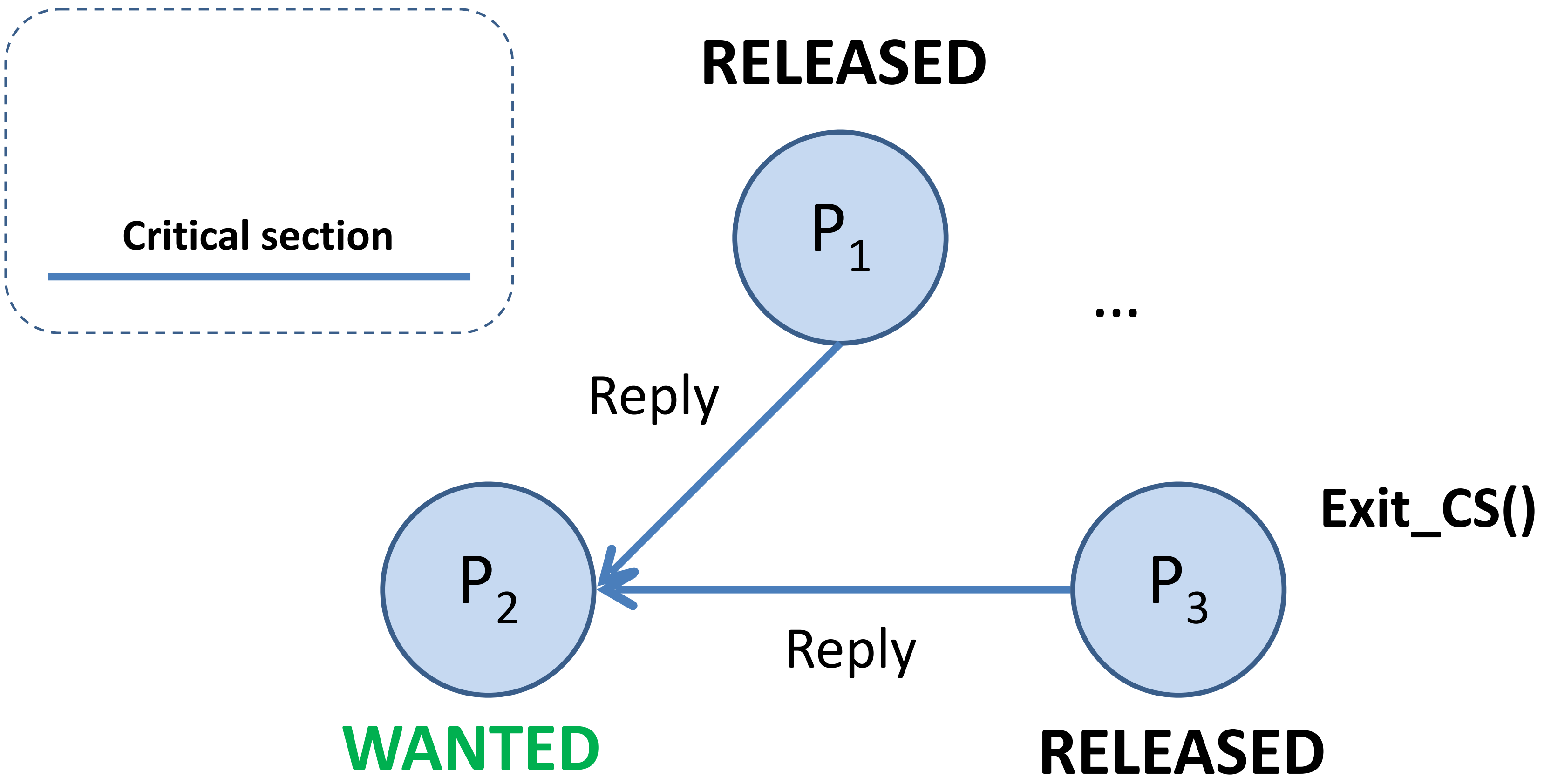
Requesting entry to CS

Request while Held



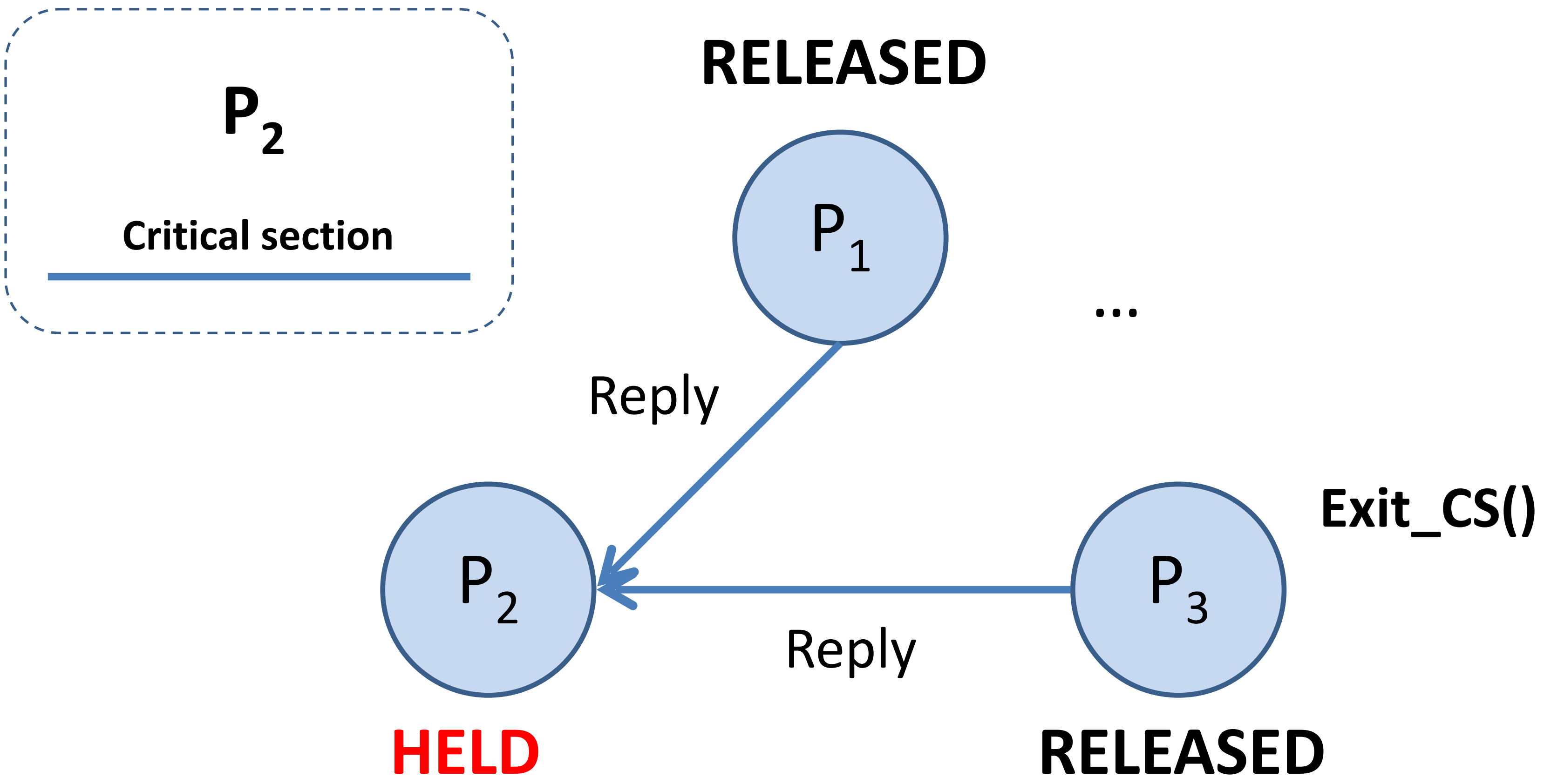
Requesting entry to CS

Request while Held



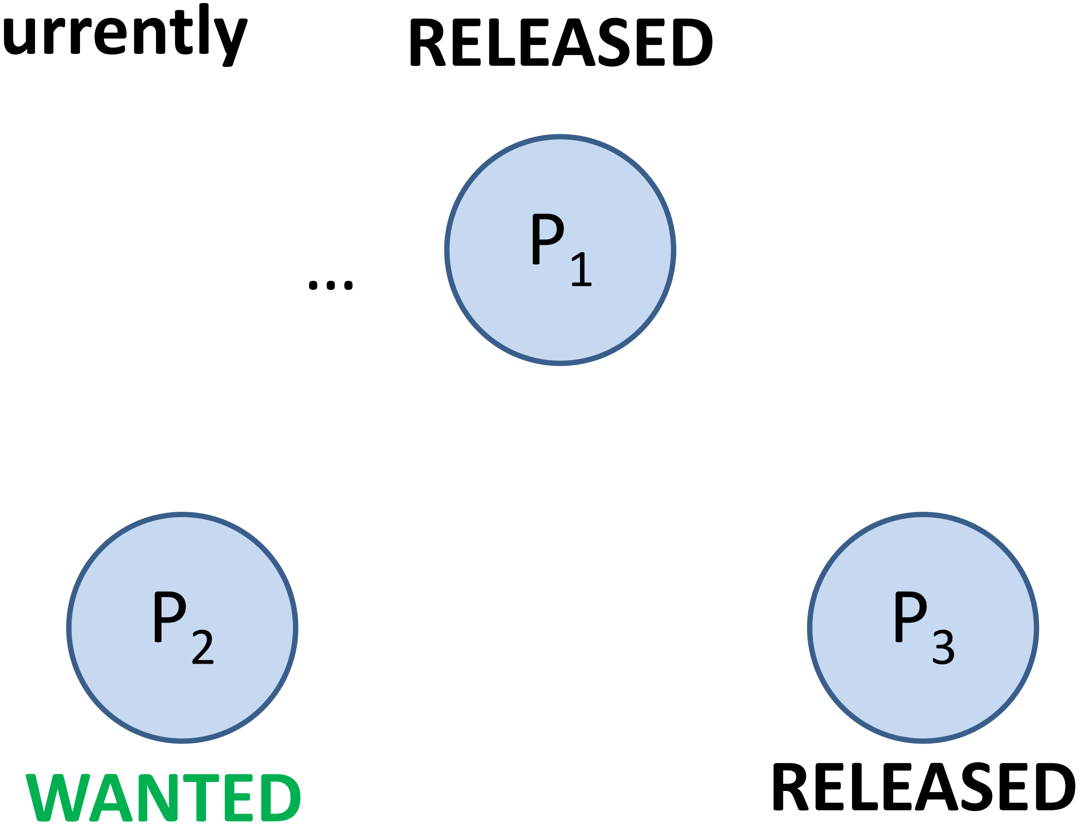
Requesting entry to CS

Request while Held



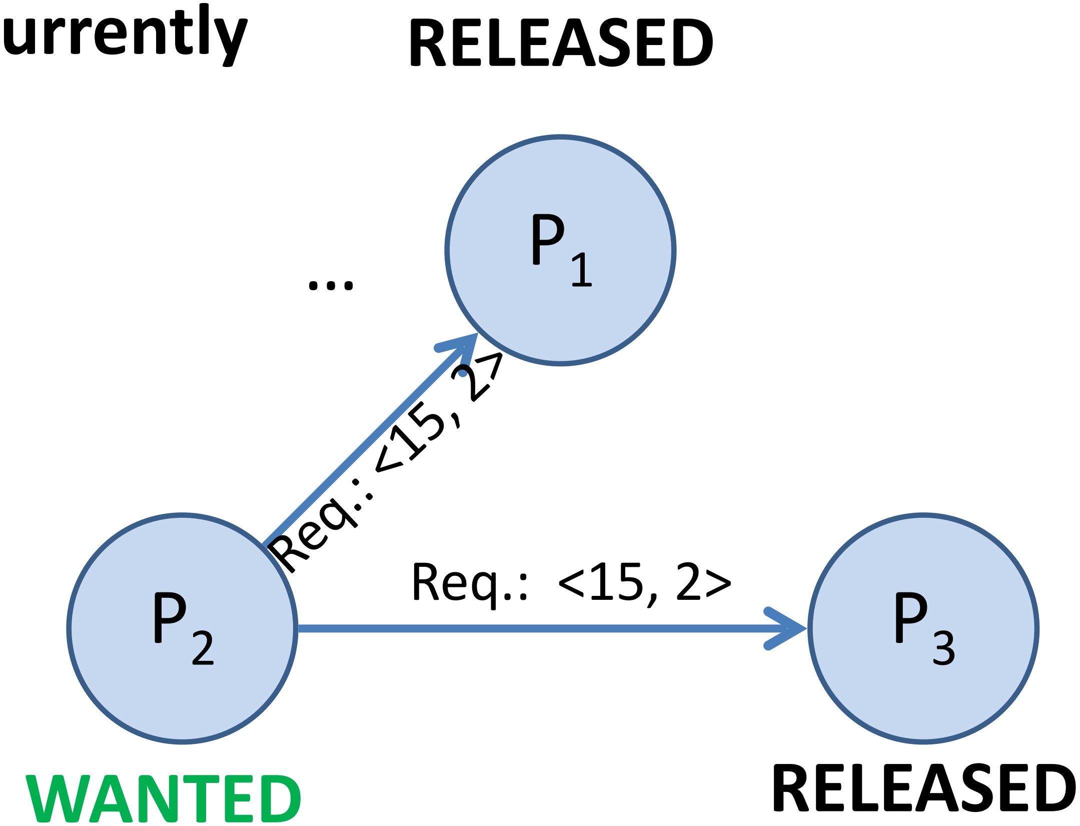
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



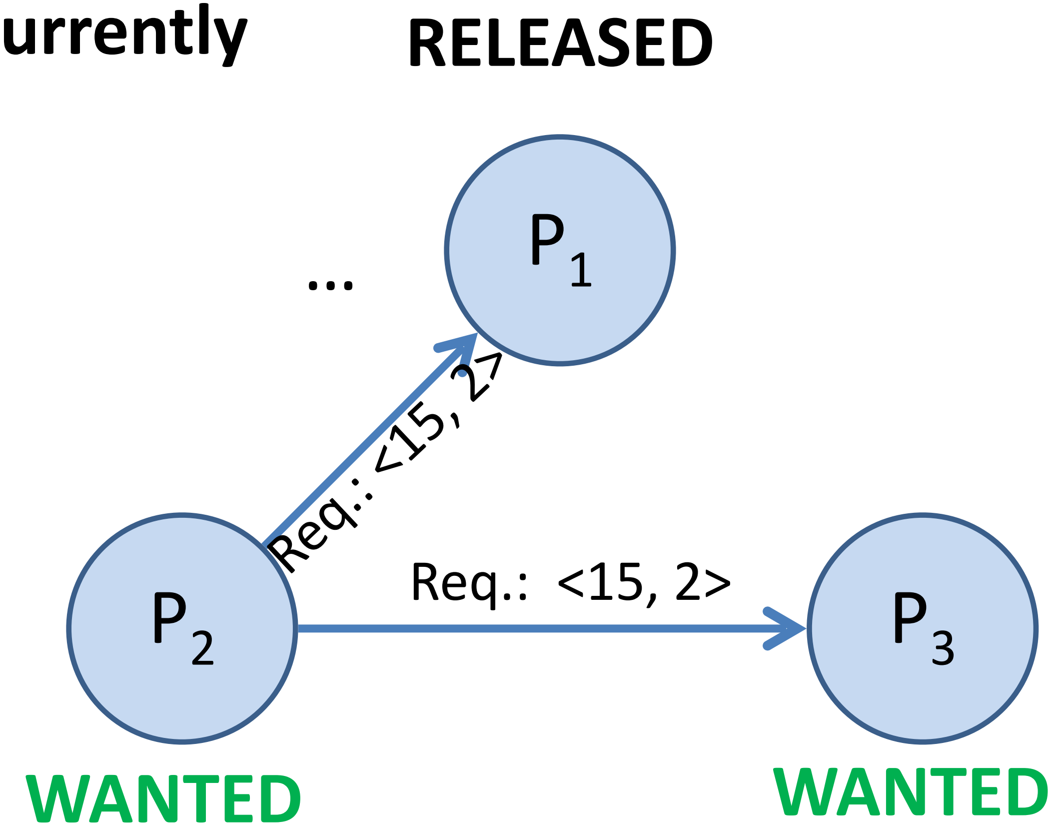
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



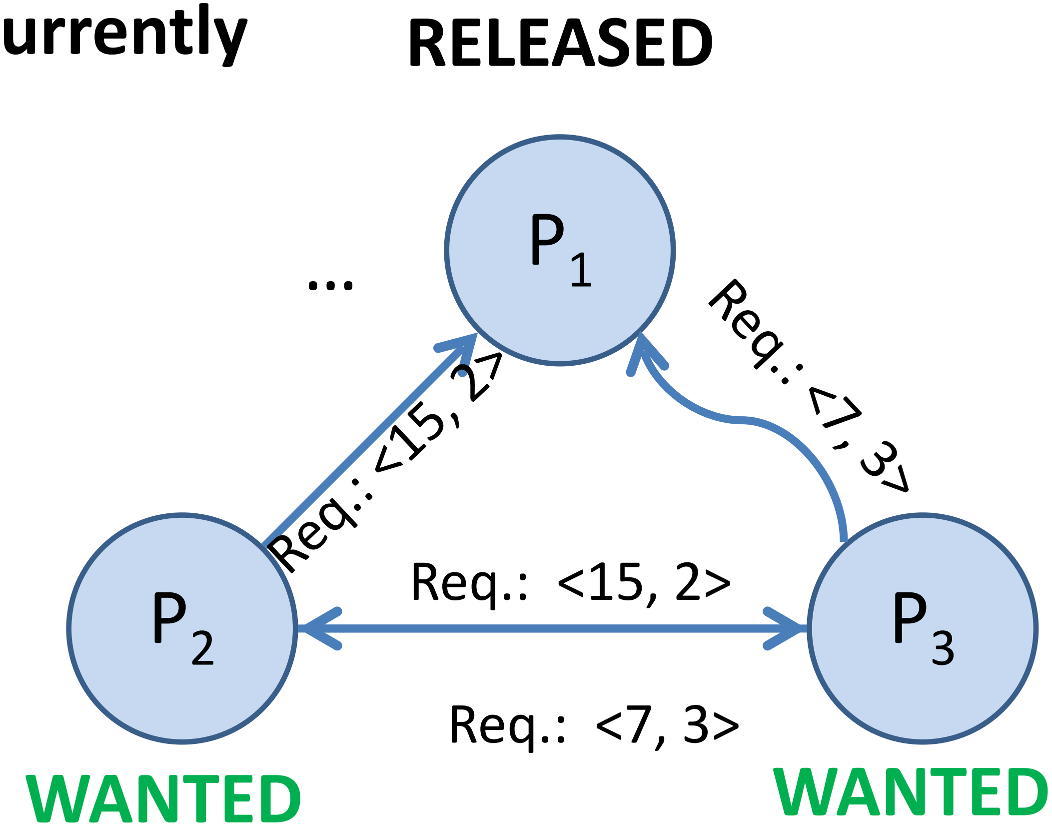
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



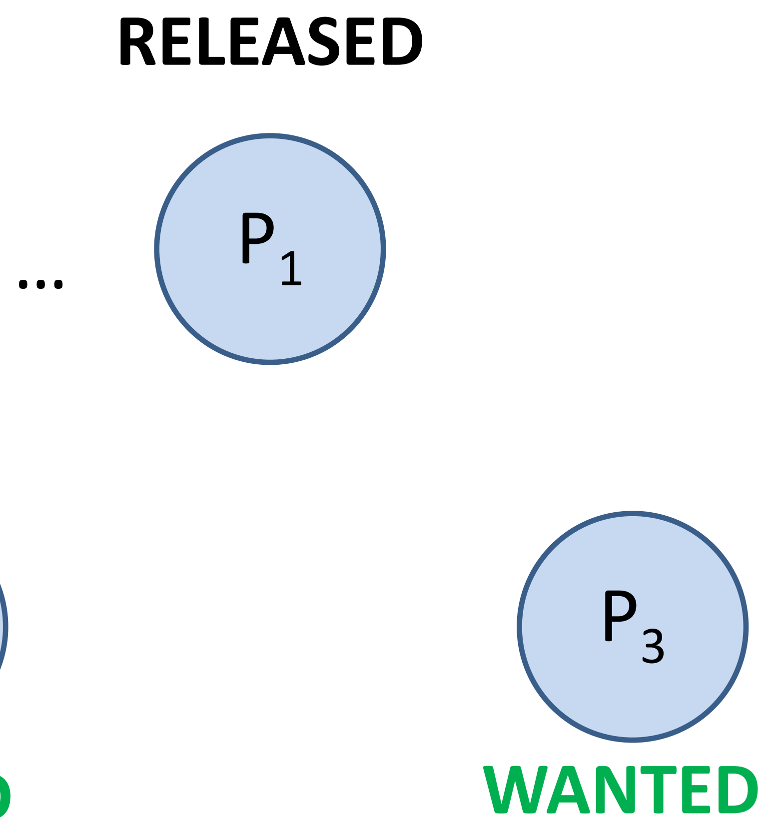
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



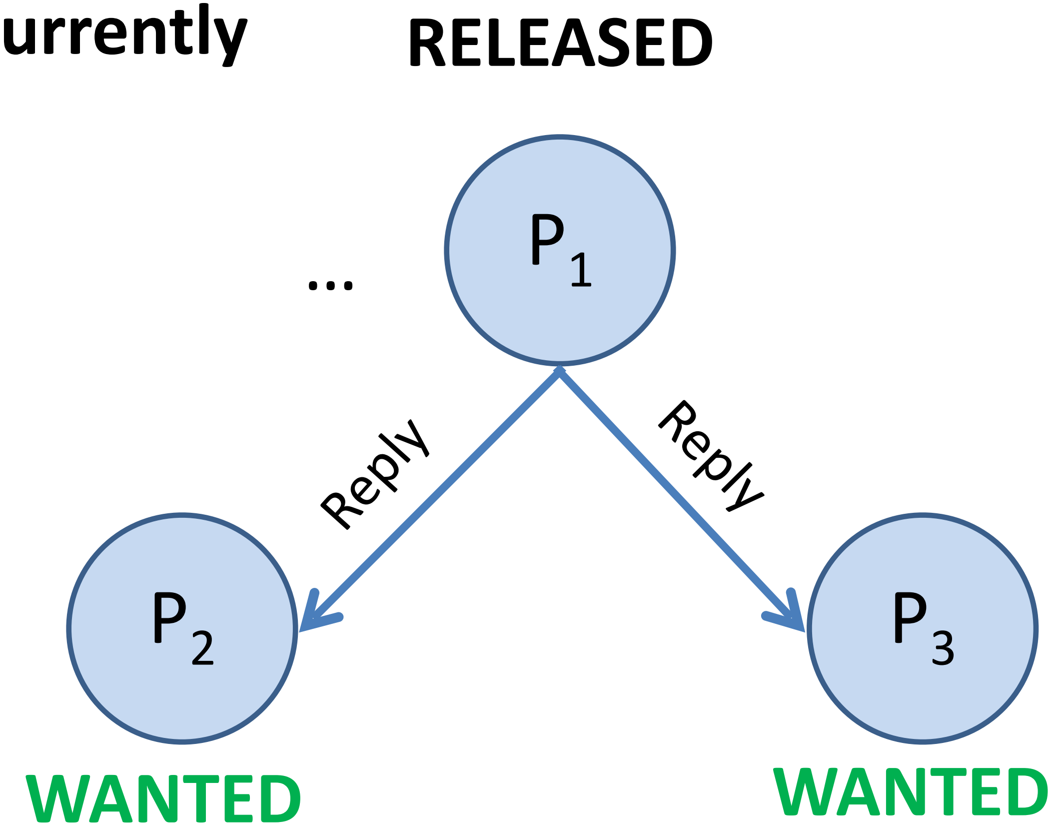
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



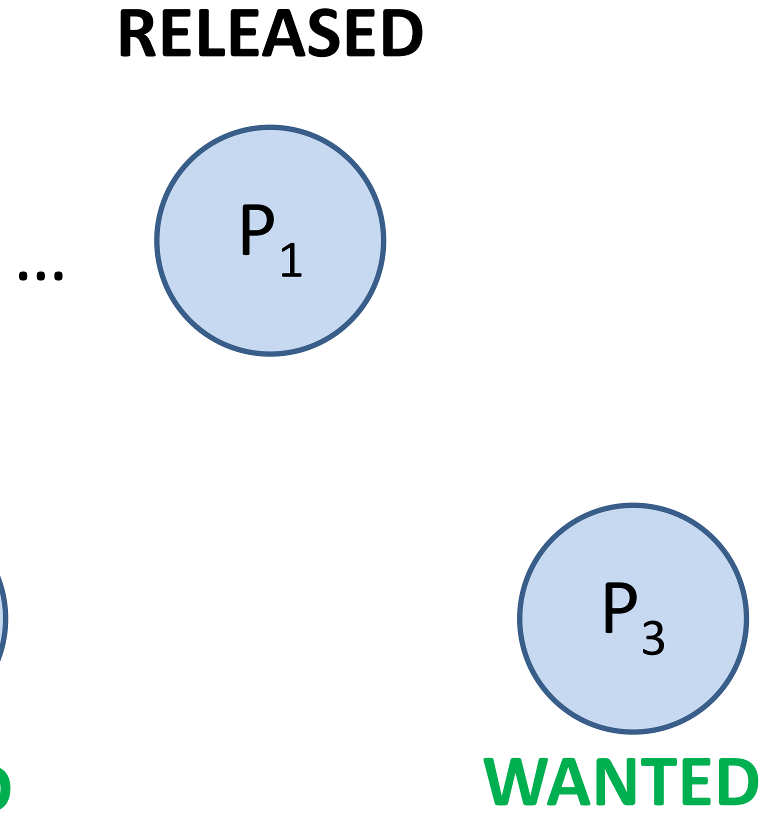
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



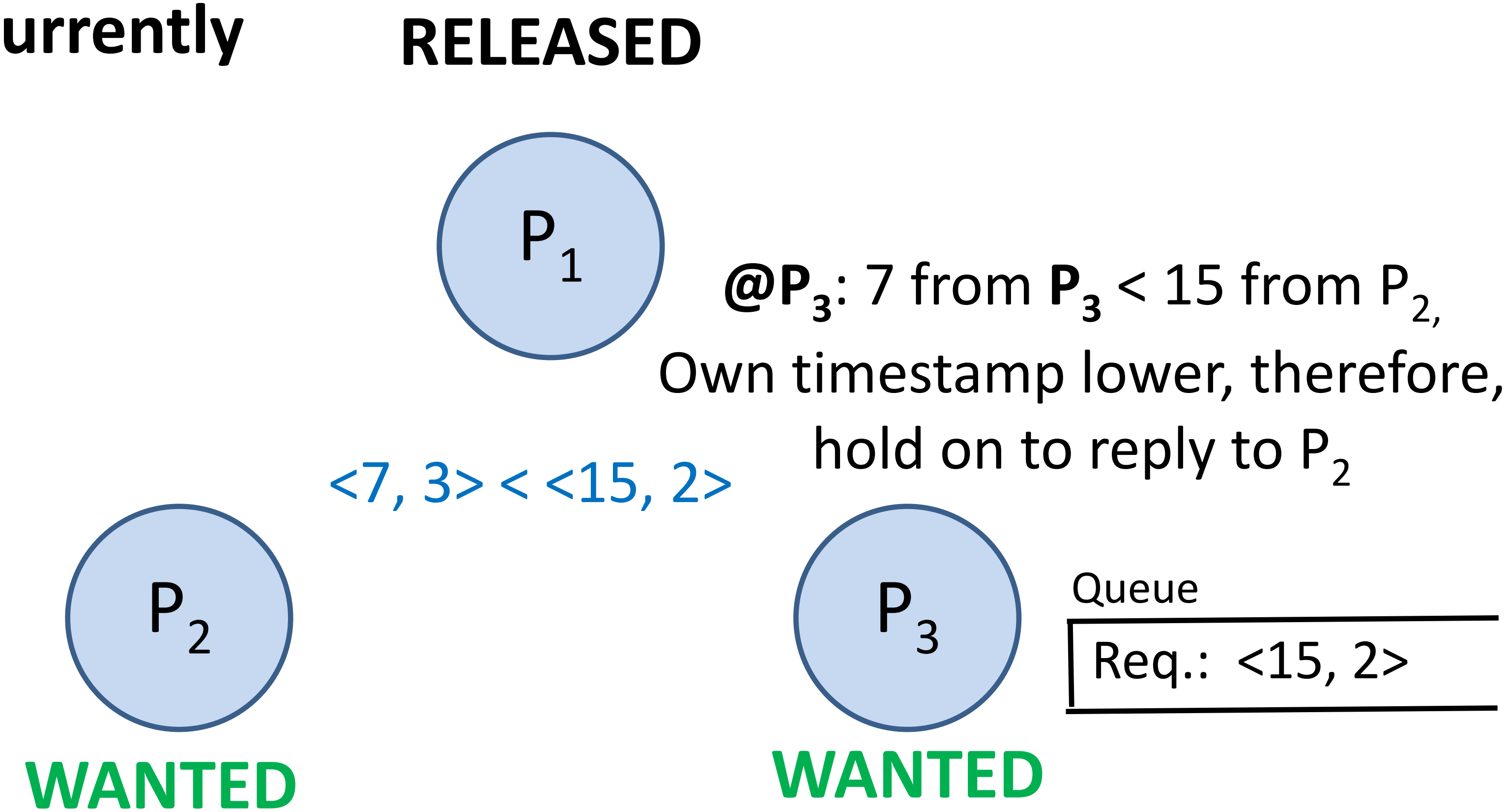
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



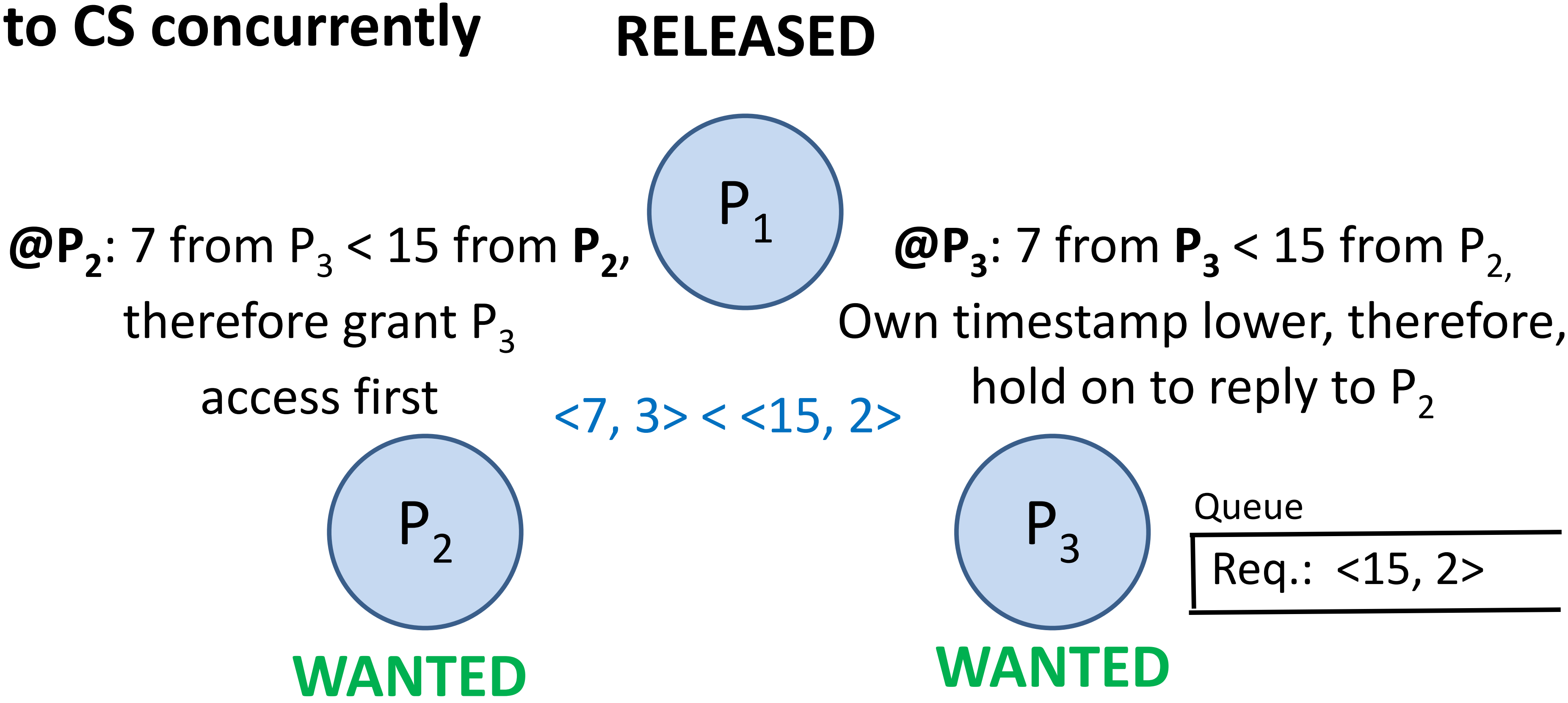
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



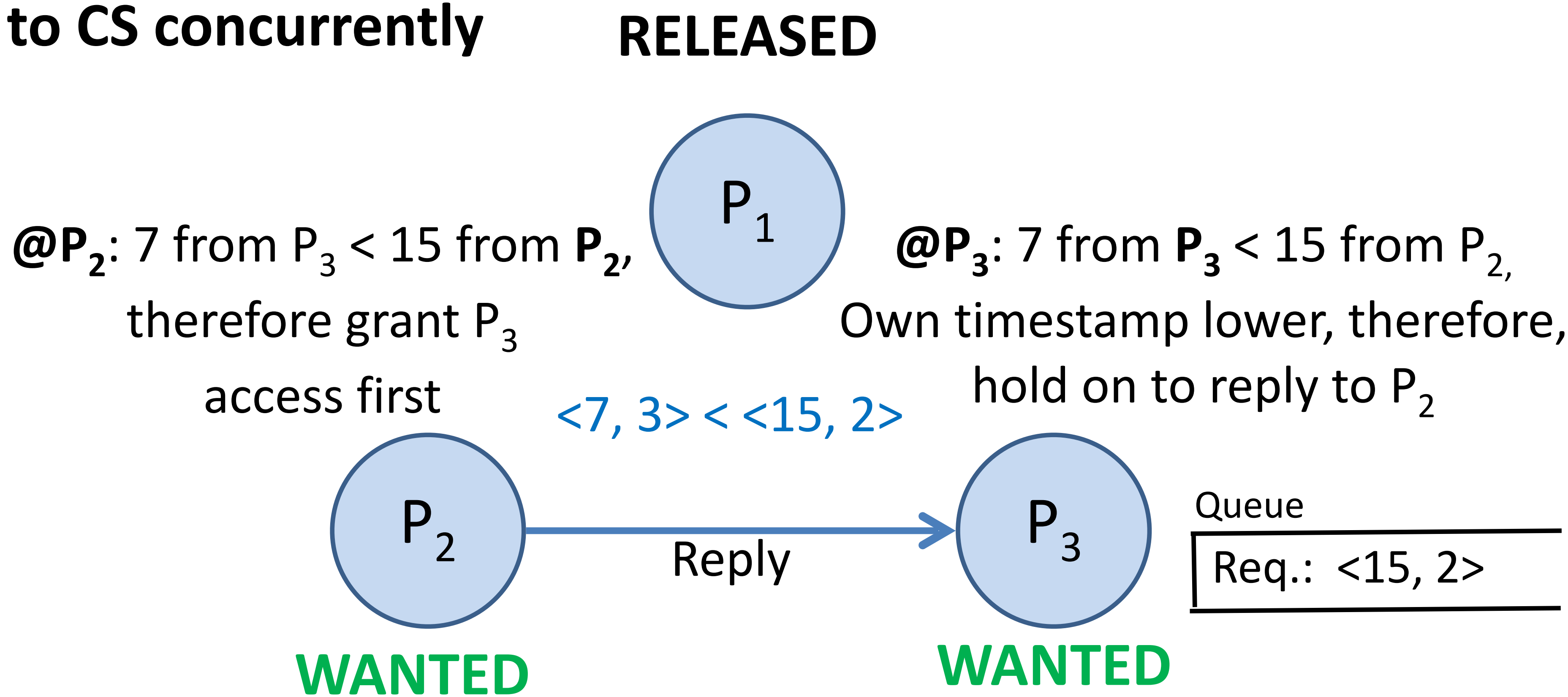
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



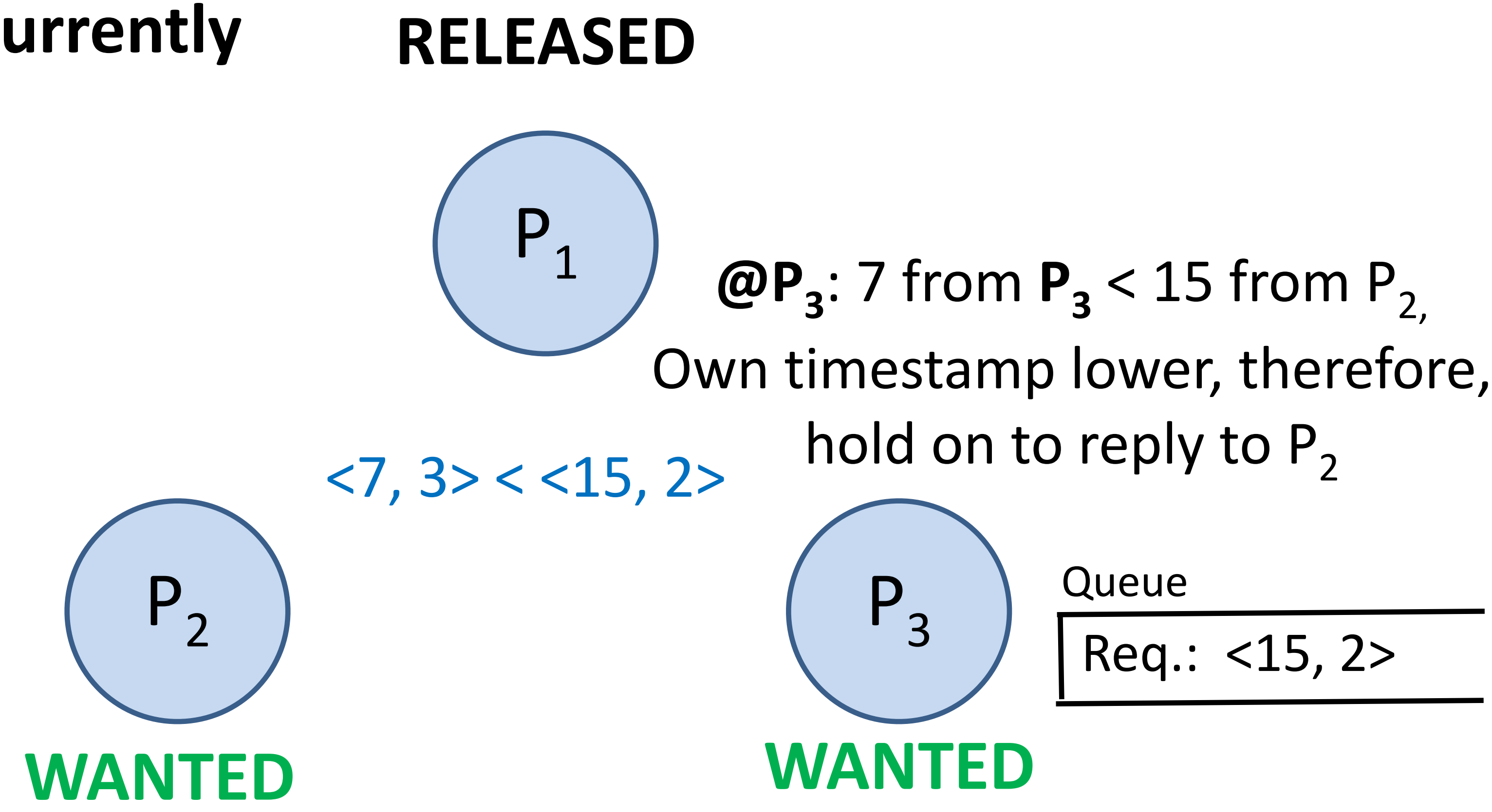
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



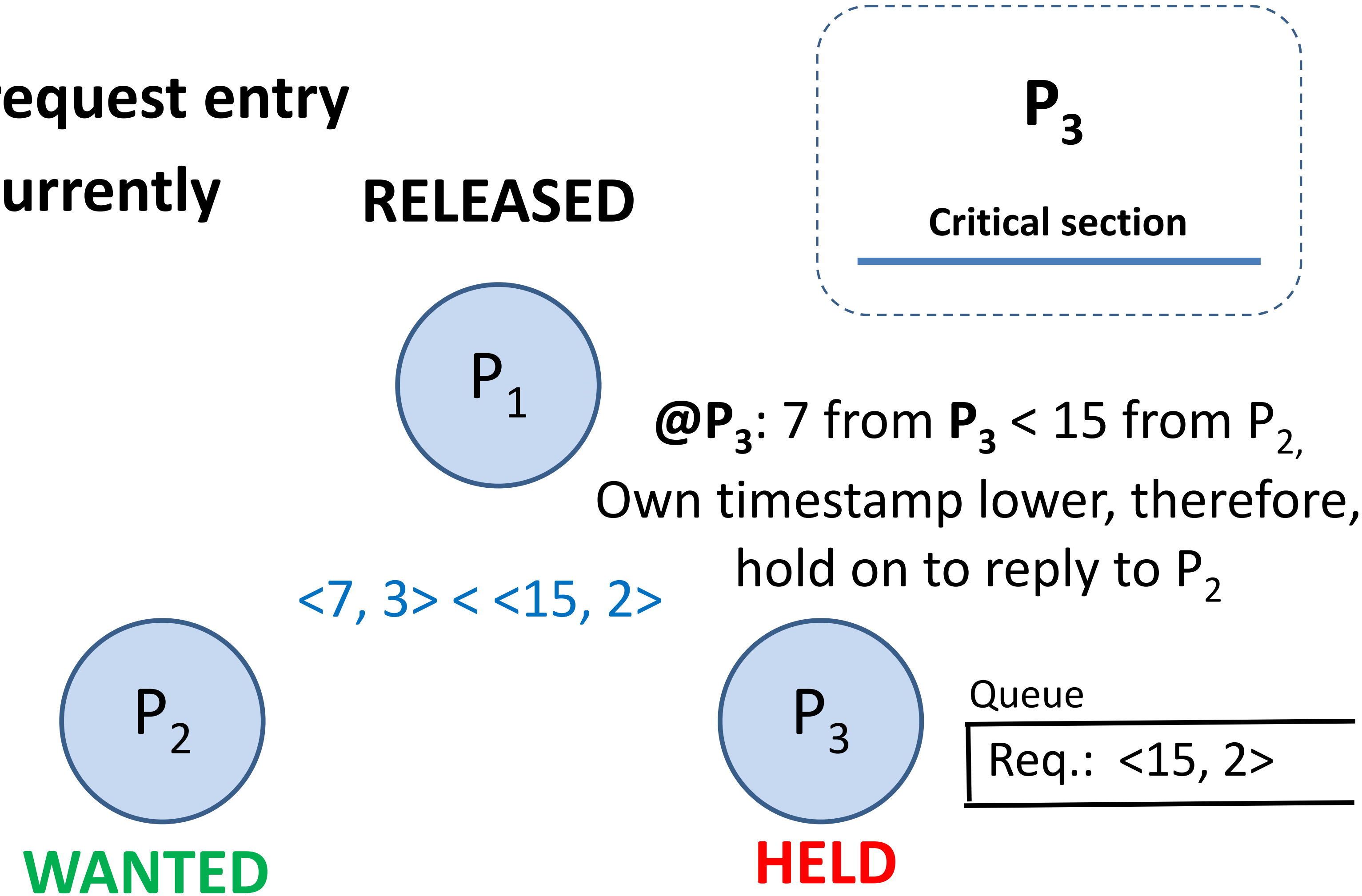
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



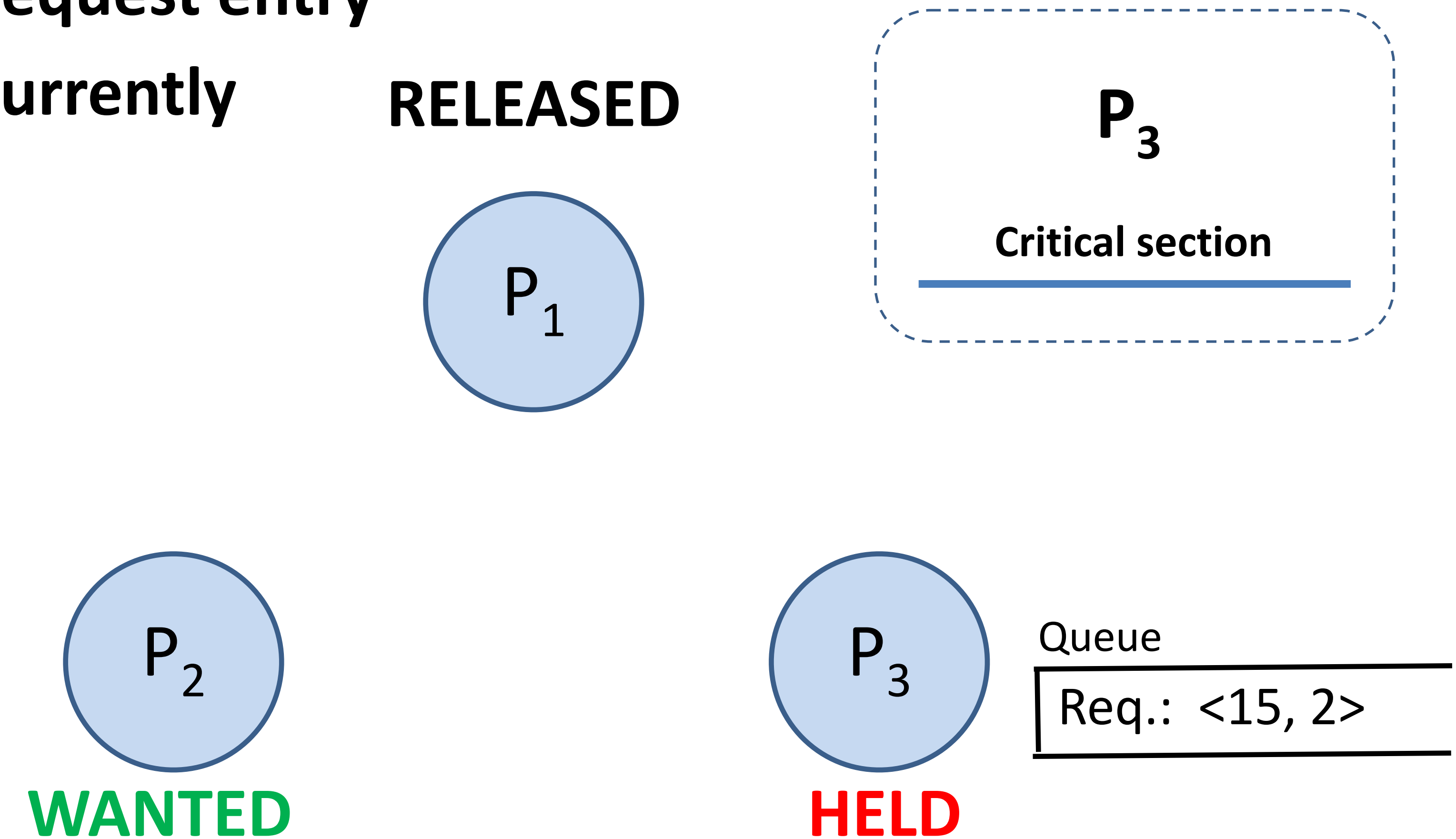
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



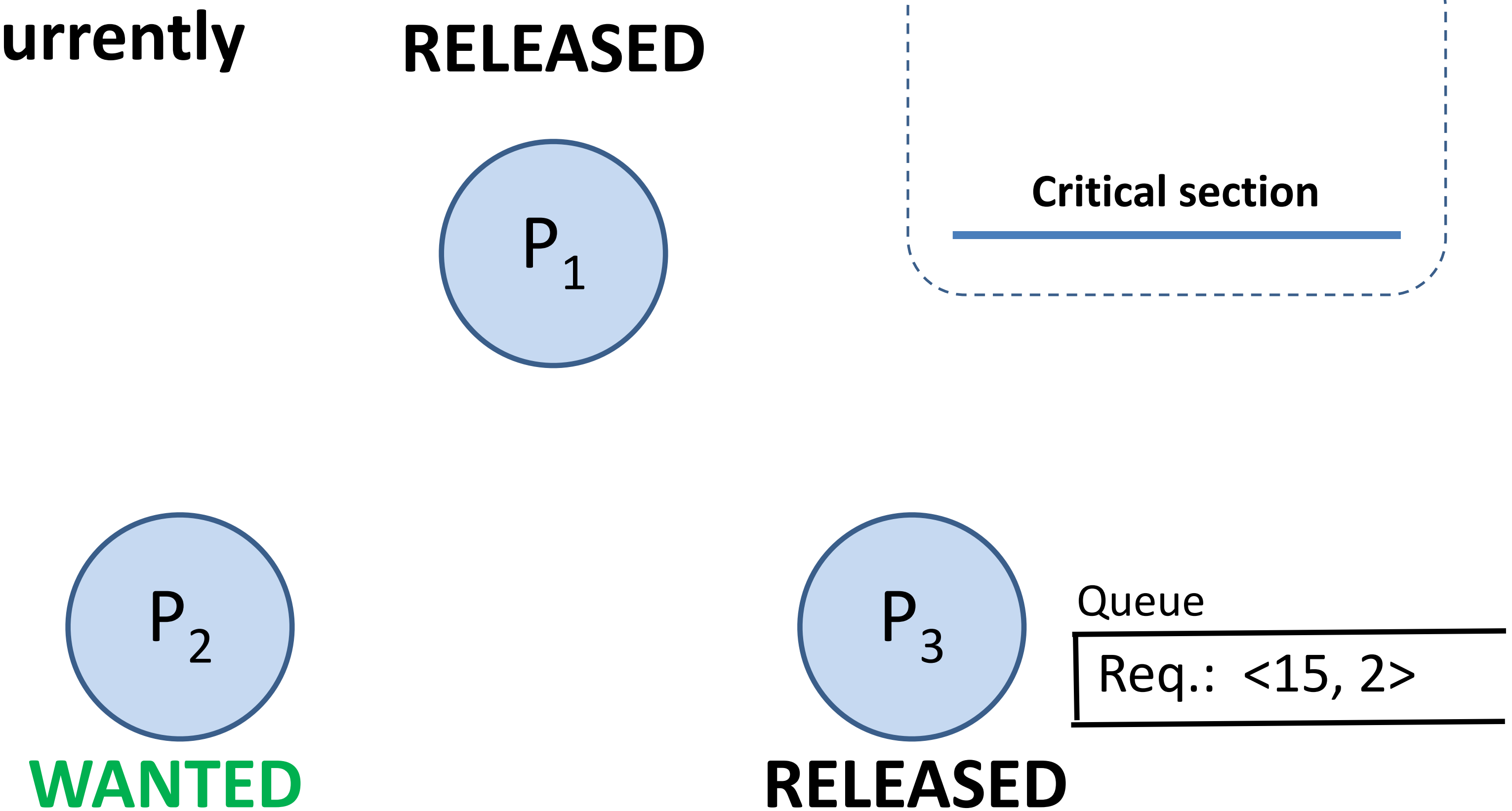
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



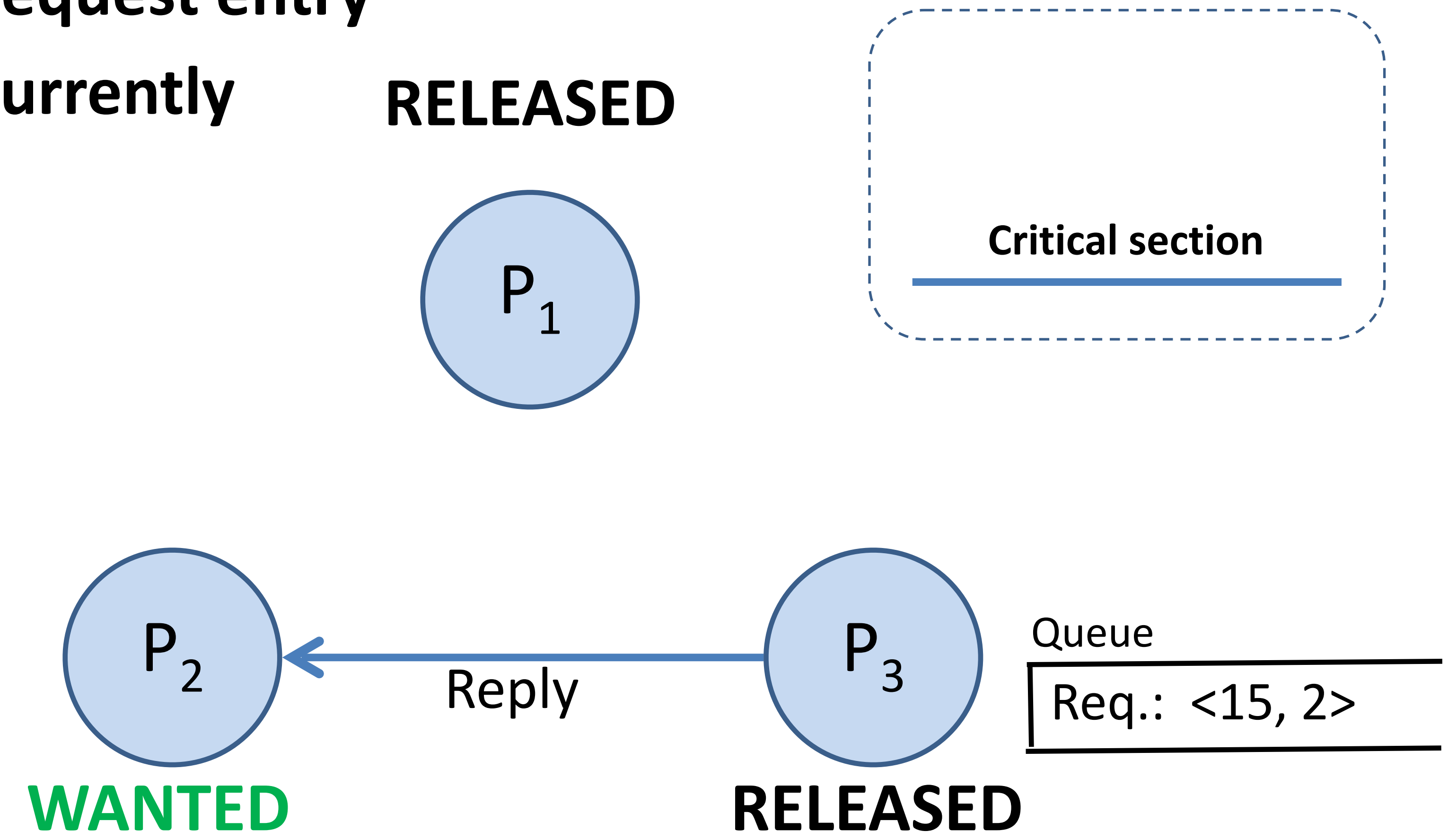
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



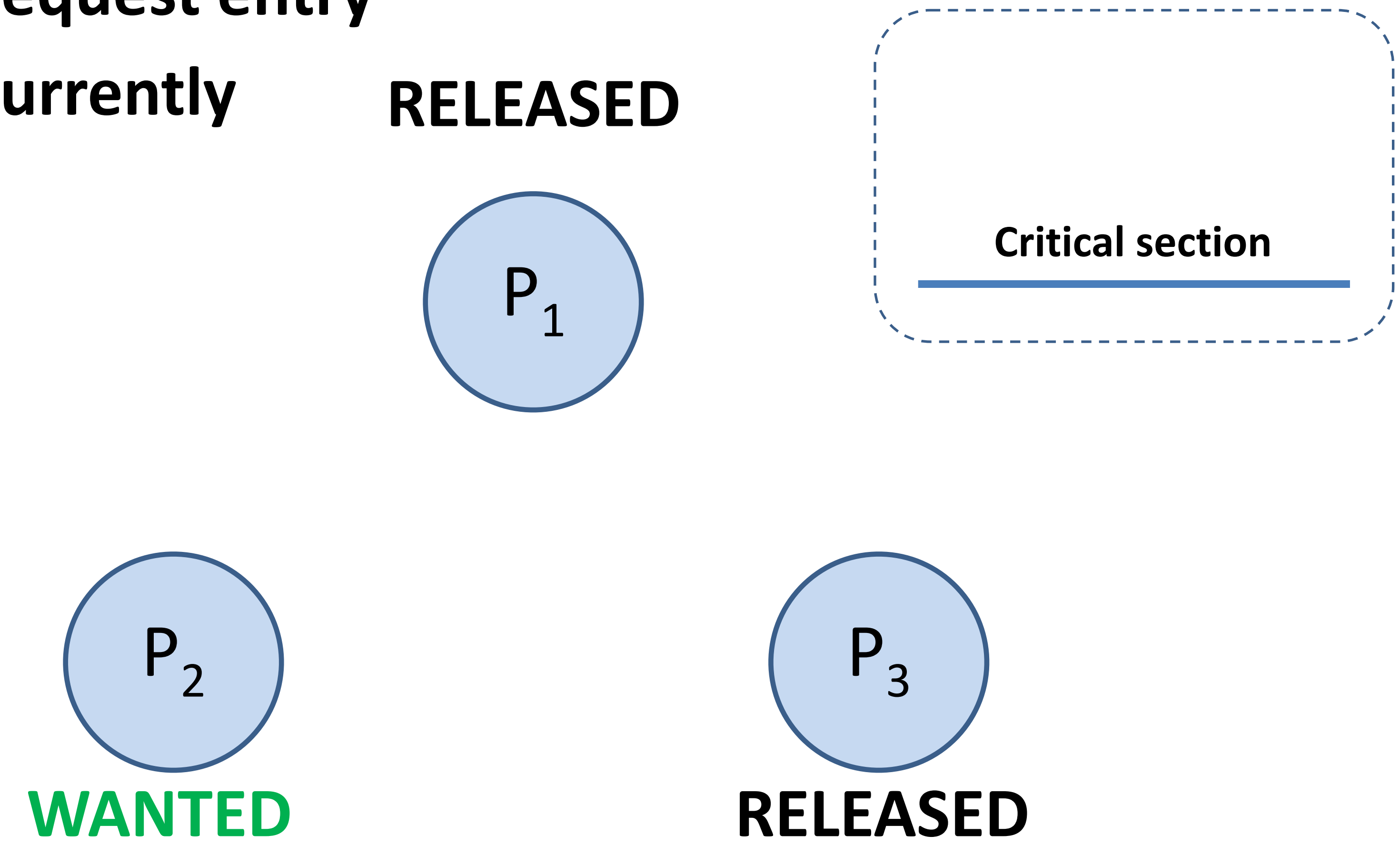
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



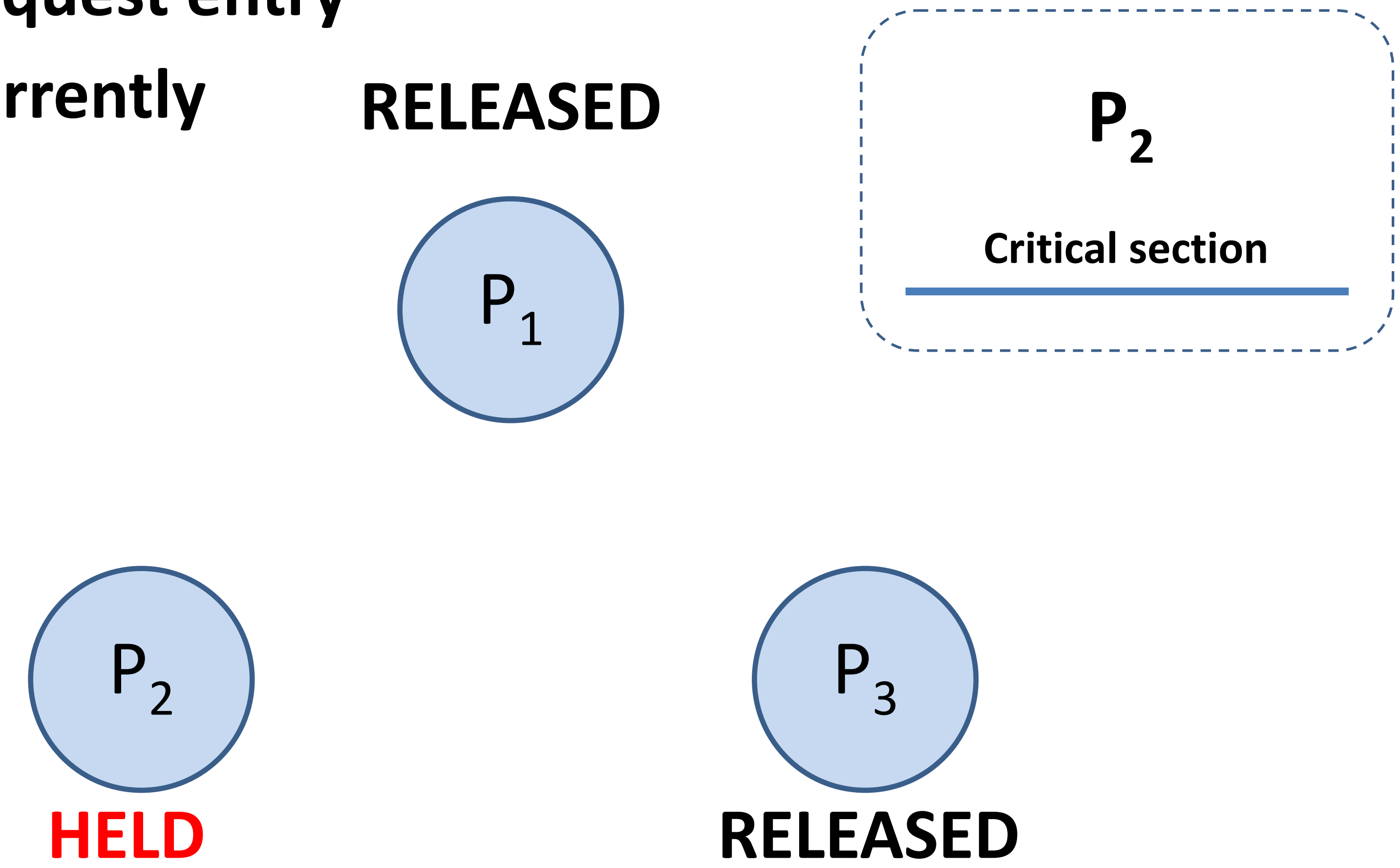
Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



Concurrent entry requests

P_2 and P_3 request entry to CS concurrently



Pseudo code

Pseudo code

On initialization

state = RELEASED

Pseudo code

```
Enter_CS()
    state = WANTED
    Broadcast timestamped
        request to all nodes
    wait until ( (n-1) acks received )
    state = HELD
```

```
On initialization
    state = RELEASED
```

Pseudo code

Enter_CS()

state = WANTED

Broadcast timestamped **request** to all nodes

wait until ((n-1) acks received)

state = HELD

On initialization

state = RELEASED

On receiving a request with ***<ts$_i$, i>*****, at P$_k$ (i ≠ k)**

if (state==HELD or (state==WANTED and ***<ts$_k$, k>*** < ***<ts$_i$, i>***))

queue request from P_i without replying

else

send a reply to P_i

k

Pseudo code

Enter_CS()

state = WANTED

Broadcast timestamped **request** to all nodes

wait until ((n-1) acks received)

state = HELD

On initialization

state = RELEASED

Exit_CS()

state = RELEASED

Reply to all queued requests

On receiving a request with <*ts_i*, *i*>, at P_k (i ≠ k)

if (state==HELD or (state==WANTED and <*ts_k*, *k*> < <*ts_i*, *i*>))

queue request from P_i without replying

else

send a reply to P_i

k

Pseudo code

Enter_CS()

state = WANTED

Broadcast timestamped **request** to all nodes

wait until ((n-1) acks received)

state = HELD

On initialization

state = RELEASED

Exit_CS()

state = RELEASED

Reply to all queued requests

***k*'s timestamp (its own)**

On receiving a request with <*ts_i*, *i*>, at P_k ($i \neq k$)

if (state==HELD or (state==WANTED and <*ts_k*, *k*> < <*ts_i*, *i*>))

queue request from P_i without replying

else

send a reply to P_i

k

Pseudo code

Enter_CS()

state = WANTED

Broadcast timestamped **request** to all nodes

wait until ((n-1) acks received)

state = HELD

On initialization

state = RELEASED

Exit_CS()

state = RELEASED

Reply to all queued requests

On receiving a request with $<ts_i, i>$**, at** P_k $(i \neq k)$

if (state==HELD or (state==WANTED and $<ts_k, k>$ < $<ts_i, i>$))

queue request from P_i without replying

else

send a reply to P_i

k

Reminder: Subtlety about timestamps

- Use **Lamport timestamps** to order requests: $<ts_i, i>$, ts_i the timestamp, i the node identifier

- If for two timestamps
  - $<ts_i, i>$ and $<ts_k, k>$, if $ts_i = ts_i$, then break ties by looking at node identifiers i, k
  - Gives rise to an **arbitrary total order** over timestamps (i.e., requests)

Summarizing observations I

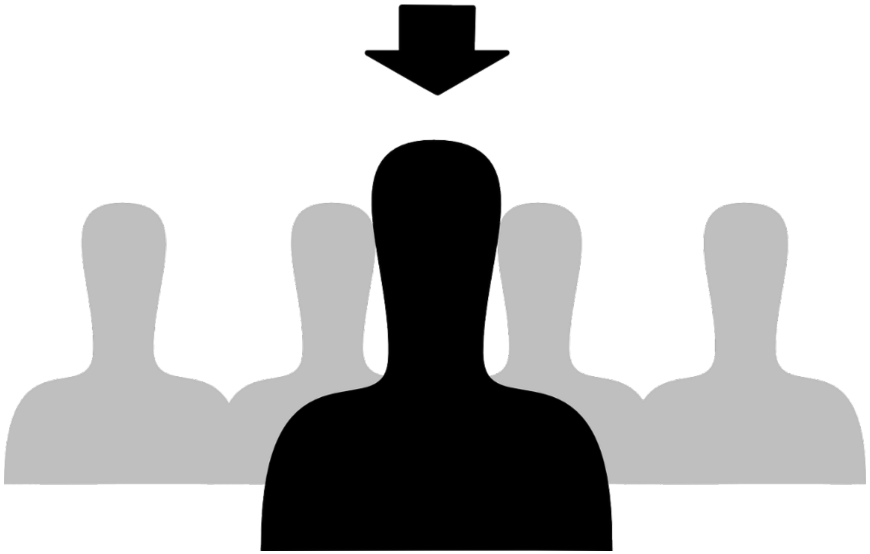
- **Safe**
  - If two nodes were in the CS at the same time, they would have had to reply to each other, which can’t happen since their requests are totally ordered
- **Live**
  - Each request is enqueued, so eventually it will be satisfied
- **Ordered**
  - Based on timestamps

Summarizing observations II

- Each entry request requires 2(N-1) messages
  - N-1 requests, N-1 replies
- Synchronization delay is one message
- Not fault tolerant

Self-study questions

- Would node IDs as timestamps suffice, why or why not?
- Would Lamport clock values as timestamps suffice, why or why not?
- Is the algorithm order-preserving, why or why not?
- Compare all our distributed mutual exclusion algorithms according to bandwidth use, synchronization delay, and delay at entry and exit.
- Compare all our distributed mutual exclusion algorithms according to their failure resilience.



Pixabay license

LEADER ELECTION

OVERVIEW

Leader election

a.k.a., coordinator, master, etc.

- **Problem**: A group of nodes, P_1, …, P_n, **must agree** on some **unique P_k** to be the “**leader**”
- Often, leader then **coordinates another activity**
- Election runs when leader failure is detected (suspected)
- Any node who hasn’t heard from leader in some predefined time interval **may call for an election**
- **False alarm** is a possibility (new elections initiated, while current leader still alive)
- **Several nodes** may initiate **elections concurrently**
- Algorithm should allow for node crash during election

Leader election use cases

- Berkeley clock synchronization algorithm
- Centralized mutual exclusion algorithm
- Leader election for choosing *master* in Hbase, Bigtable
- Choosing *master* among *n* nodes in Chubby or ZooKeeper coordination service

- *Primary-backup replication algorithms*
- *Two-phase commit protocol*

Leader election vs. mutual exclusion

- Election losers return to what they were doing ...

  ... **instead of waiting**
- **Fast election** is important ...

  ... not starvation avoidance
- **All nodes** must know result ...

  ... not just the winner

Leader election vs. mutual exclusion

- Election losers return to what they were doing ...

  ... **instead of waiting**
- **Fast election** is important ...

  ... not starvation avoidance
- **All nodes** must know result ...

  ... not just the winner

ME can be reduced to LE!
(e.g., HBase wants LE,
ZooKeeper provides ME)

Uniqueness requirement

Unique identifier

- Elected **leader** must be **unique**
- Active nodes with **largest identifier** wins
- Unique identifier (UID) could be any "useful value"
  - I.e., **unique** and **totally orderable value**
- E.g., based on process identifiers, IP-port-number
- E.g., based on least computational load
  - $<1/load, i>$, $load > 0$, i is UID to break ties
- Each node, $\boldsymbol{P_i}$, has a variable $\boldsymbol{elected_i}$ that holds the value of the leader or is "$\perp$" (undefined)

Election algorithm requirement

- **Safety** - correctness
  - A participating node $\boldsymbol{P_i}$ has variable $\boldsymbol{elected_i}$ **= "$\perp$"** or $\boldsymbol{elected_i}$ **= $\boldsymbol{P}$,** where $\boldsymbol{P}$ is chosen as the non-crashed node at the end of the election run with the **largest identifier.**
  - **Only one leader at a time!**

- **Liveness** – progress is made
  - **All nodes participate** in the election and **eventually** either set $\boldsymbol{elected_i} \neq$ **"$\perp$"** or **crash**.

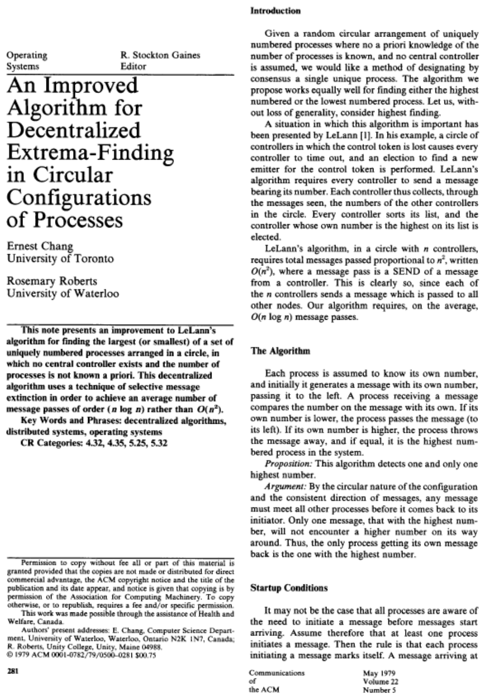
Summary

- Leader election is a fundamental building block in designing distributed systems

- Leader election algorithms overview (a popular shortlist)
  – Chang & Roberts, 1979
  – HS algorithm, 1980
  – Bully, 1982
  – Leader Election in Raft, 2014

Self-study questions

- Why settle on the largest unique identifier as determining characteristic for leader?
- Develop your own leader election algorithm by reducing the problem to mutual exclusion.
- Find a few more leader election use case scenarios.



Operating Systems — R. Stockton Gaines, Editor

An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes

Ernest Chang
University of Toronto

Rosemary Roberts
University of Waterloo

This note presents an improvement to LeLann's algorithm for finding the largest (or smallest) of a set of uniquely numbered processes arranged in a circle, in which no central controller exists and the number of processes is not known a priori. This decentralized algorithm uses a technique of selective message extinction in order to achieve an average number of message passes of order (n log n) rather than $O(n^2)$.

Key Words and Phrases: decentralized algorithms, distributed systems, operating systems

CR Categories: 4.32, 4.35, 5.25, 5.32

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

This work was made possible through the assistance of Health and Welfare, Canada.

Authors' present addresses: E. Chang, Computer Science Department, University of Waterloo, Waterloo, Ontario N2K 1N7, Canada; R. Roberts, Unity College, Unity, Maine 04988.

© 1979 ACM 0001-0782/79/0500-0281 $00.75

Introduction

Given a random circular arrangement of uniquely numbered processes where no a priori knowledge of the number of processes is known, and no central controller is assumed, we would like a method of designating by consensus a single unique process. The algorithm we propose works equally well for finding either the highest numbered or the lowest numbered process. Let us, without loss of generality, consider highest finding.

A situation in which this algorithm is important has been presented by LeLann [1]. In his example, a circle of controllers in which the control token is lost causes every controller to time out, and an election to find a new emitter for the control token is performed. LeLann's algorithm requires every controller to send a message bearing its number. Each controller thus collects, through the messages seen, the numbers of the other controllers in the circle. Every controller sorts its list, and the controller whose own number is the highest on its list is elected.

LeLann's algorithm, in a circle with n controllers, requires total messages passed proportional to n^2, written $O(n^2)$, where a message pass is a SEND of a message from a controller. This is clearly so, since each of the n controllers sends a message which is passed to all other nodes. Our algorithm requires, on the average, $O(n \log n)$ message passes.

The Algorithm

Each process is assumed to know its own number, and initially it generates a message with its own number, passing it to the left. A process receiving a message compares the number on the message with its own. If its own number is lower, the process passes the message (to its left). If its own number is higher, the process throws the message away, and if equal, it is the highest numbered process in the system.

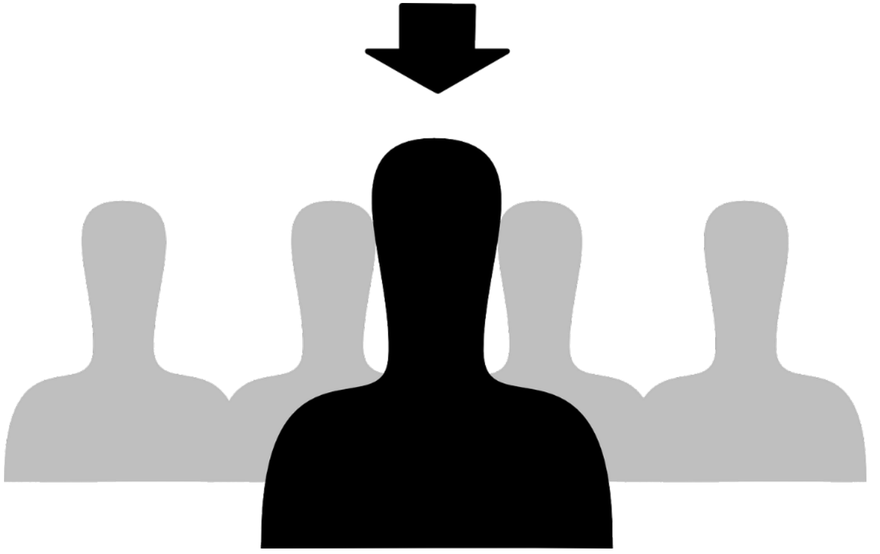
Proposition: This algorithm detects one and only one highest number.

Argument: By the circular nature of the configuration and the consistent direction of messages, any message must meet all other processes before it comes back to its initiator. Only one message, that with the highest number, will not encounter a higher number on its way around. Thus, the only process getting its own message back is the one with the highest number.

Startup Conditions

It may not be the case that all processes are aware of the need to initiate a message before messages start arriving. Assume therefore that at least one process initiates a message. Then the rule is that each process initiating a message marks itself. A message arriving at

281 — Communications of the ACM — May 1979, Volume 22, Number 5



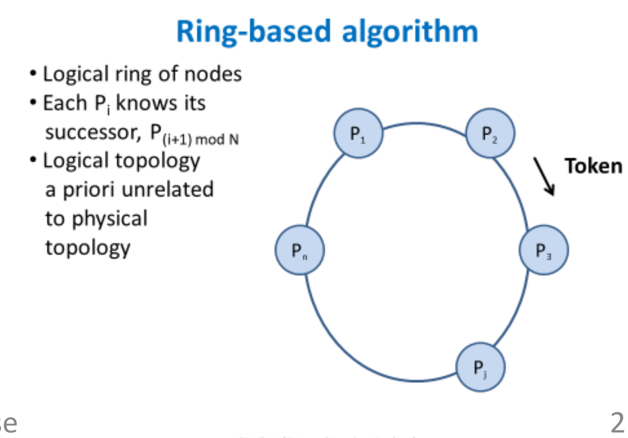
Pixabay license

LEADER ELECTION

CHANG & ROBERTS – RING-BASED ALGORITHM, 1979

Assumptions and setup

- Construct a ring (cf. ring-based mutual exclusion)
- Assume, each *P* has **unique identifier (UID)**
- Assume, no failures and asynchronous system, **but failures do happen before the election!**
- Any $\boldsymbol{P_i}$ may begin an election by sending an **election message** to its successor (i.e., after suspecting a leader failure)
- Election message holds P_i's UID



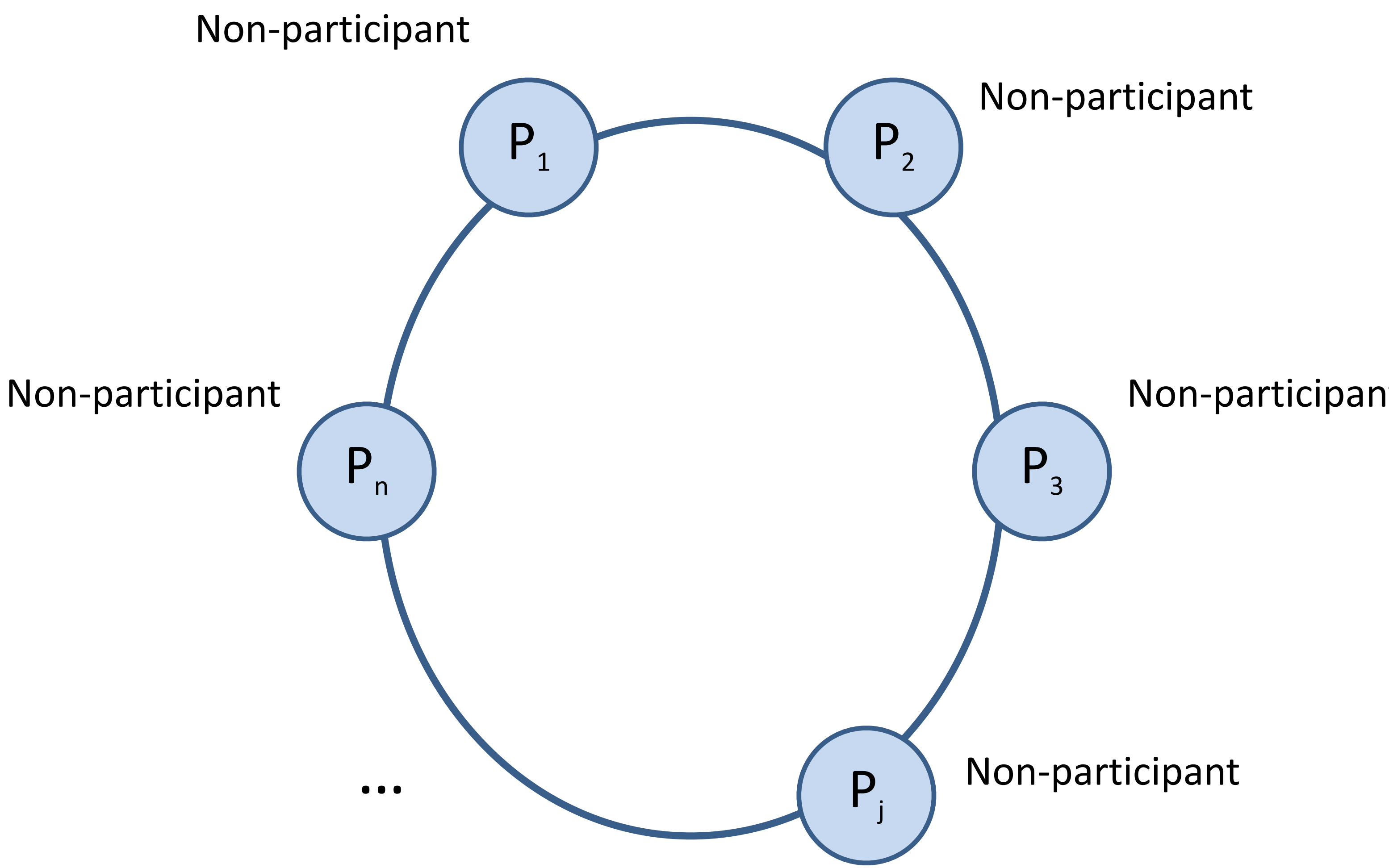
@ P_k

Ring-based election algorithm

- Election message holds P_i's UID_i
- Nodes manage a flag **PARTICIPANT** to remember their election participation status (initially false)
- Upon receiving an election message, P_k **compares its own UID_k to UID_i**
  - If $UID_i > UID_k$: **Forwards** election message
  - If $UID_i < UID_k$: **Forwards** election message **with P_k's** UID_k unless P_k has already sent a message, i.e., has participated in current election run (in that case, it does nothing)
  - If $UID_i = UID_k$: **P_k is now leader**, forwards ***victory message*** to notify all other nodes (resets PARTICIPANT flag)

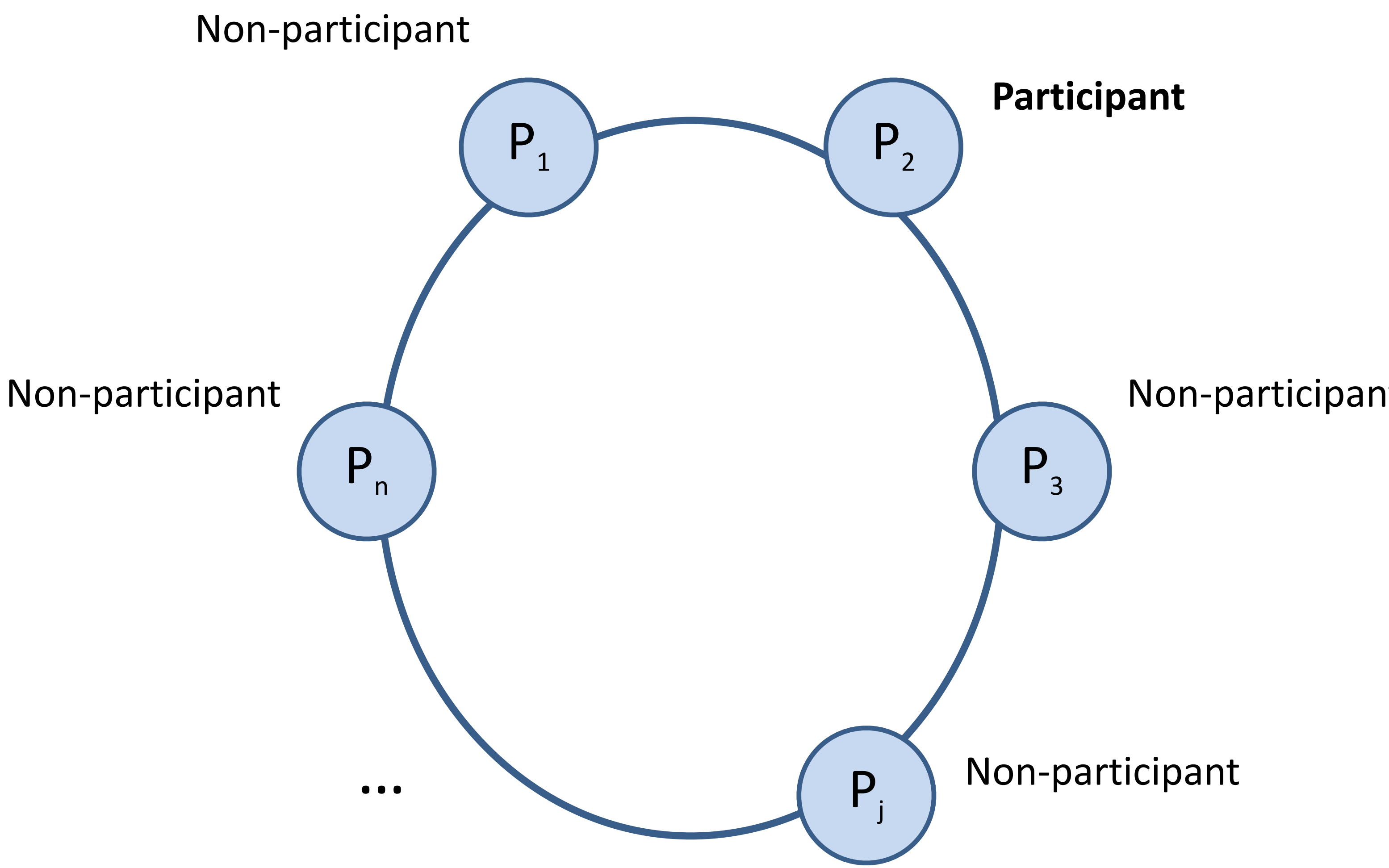
Ring-based algorithm

Calling an election (determine winner)



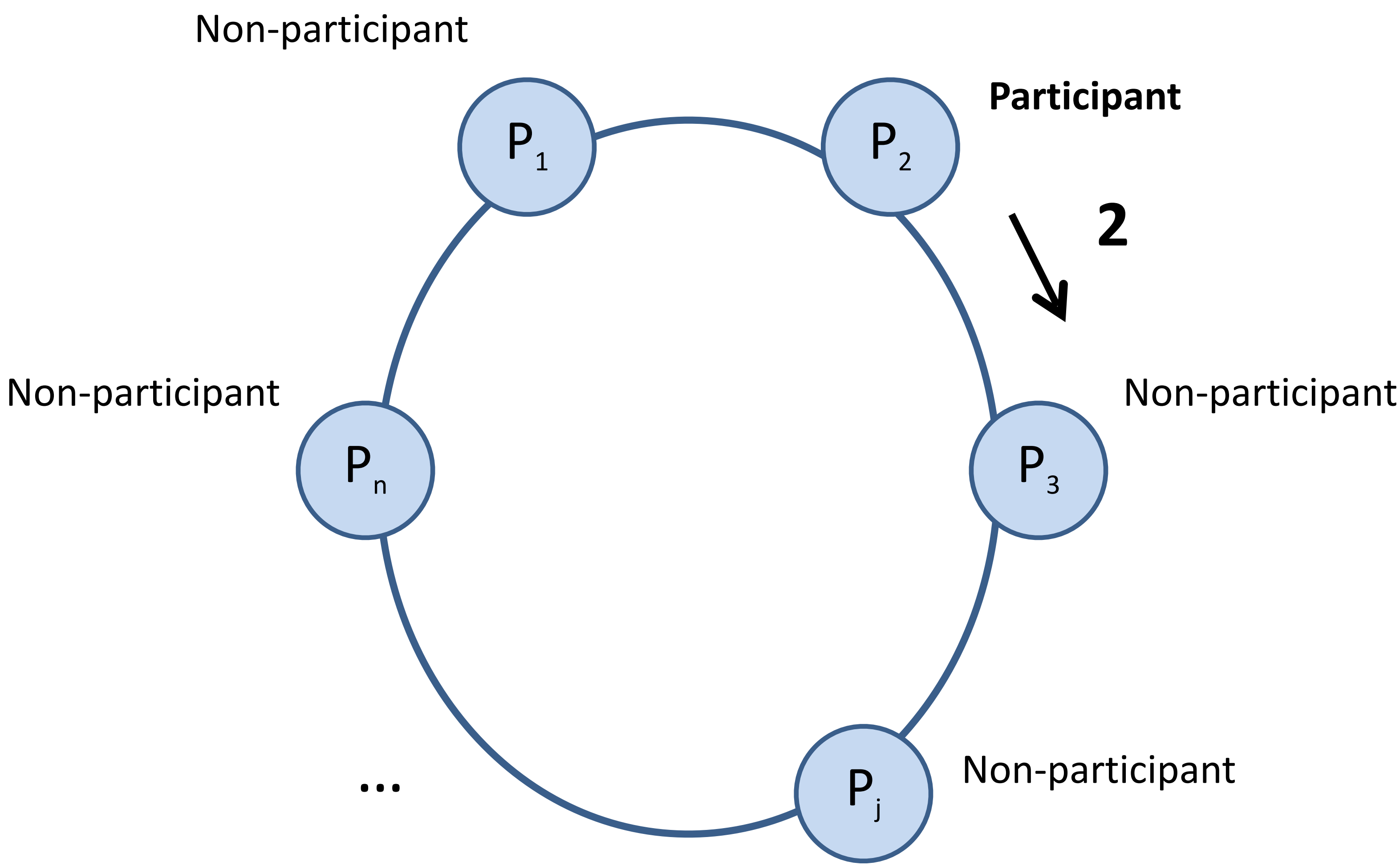
Ring-based algorithm

Calling an election (determine winner)



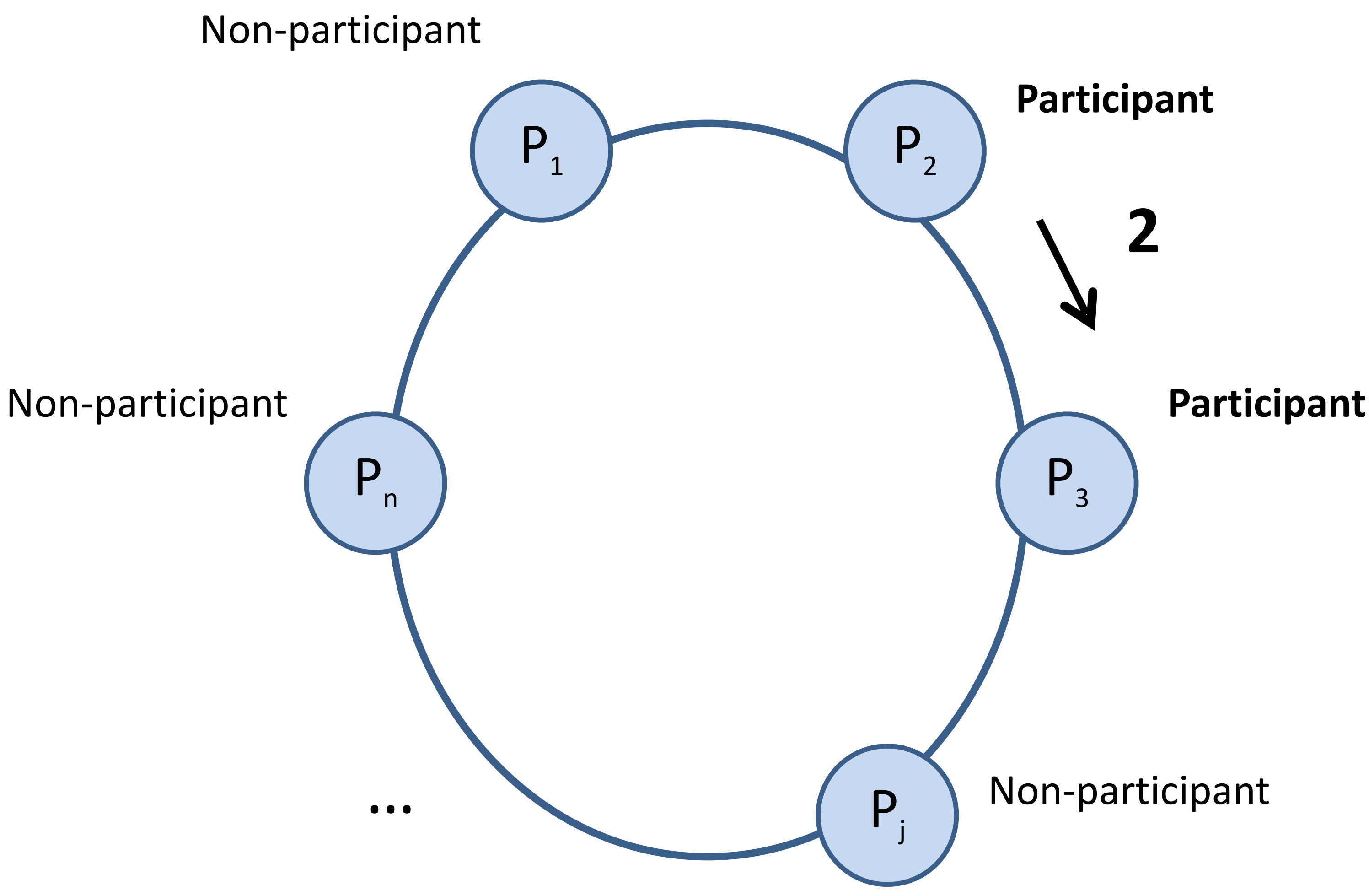
Ring-based algorithm

Calling an election (determine winner)



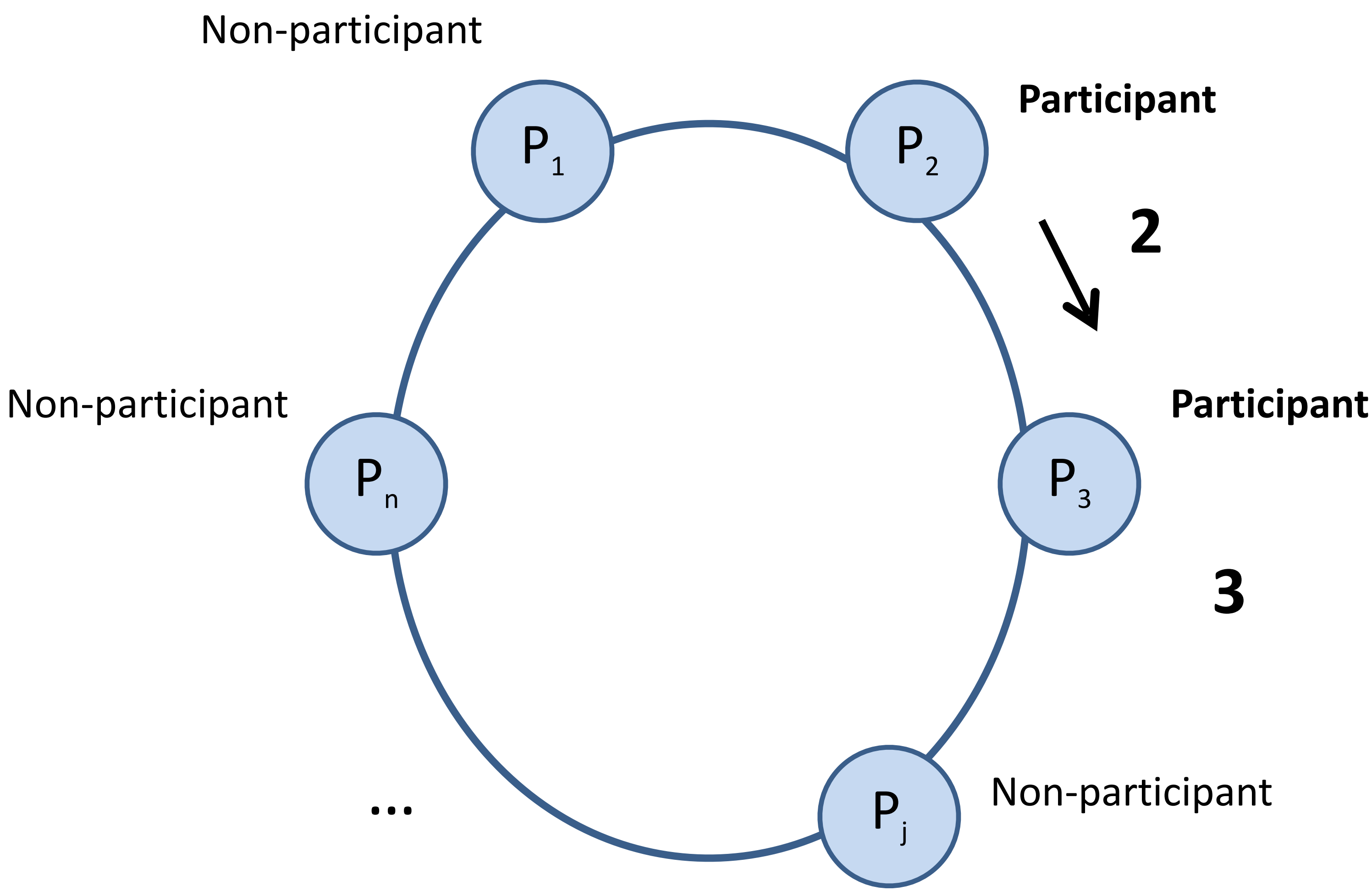
Ring-based algorithm

Calling an election (determine winner)



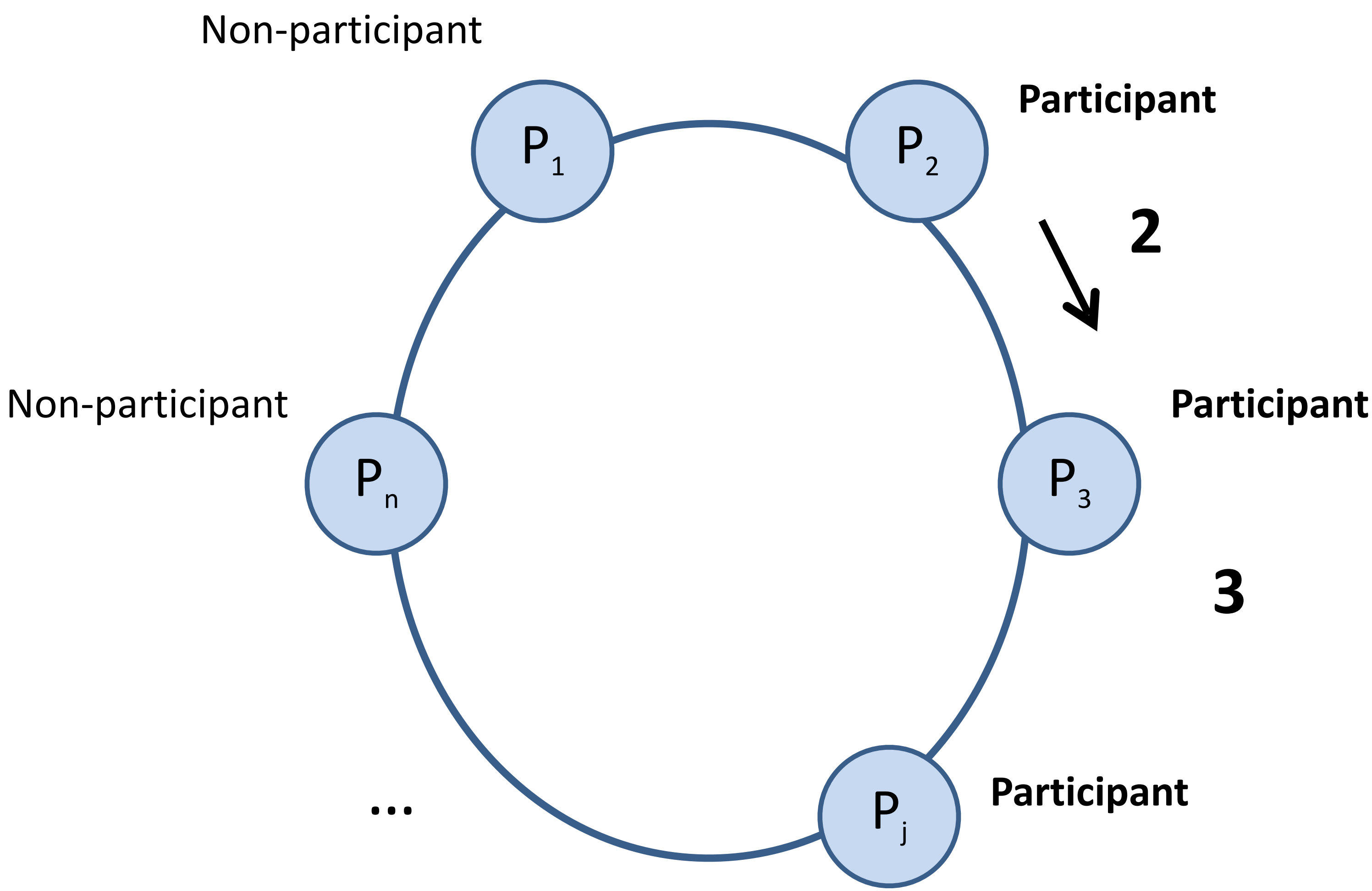
Ring-based algorithm

Calling an election (determine winner)



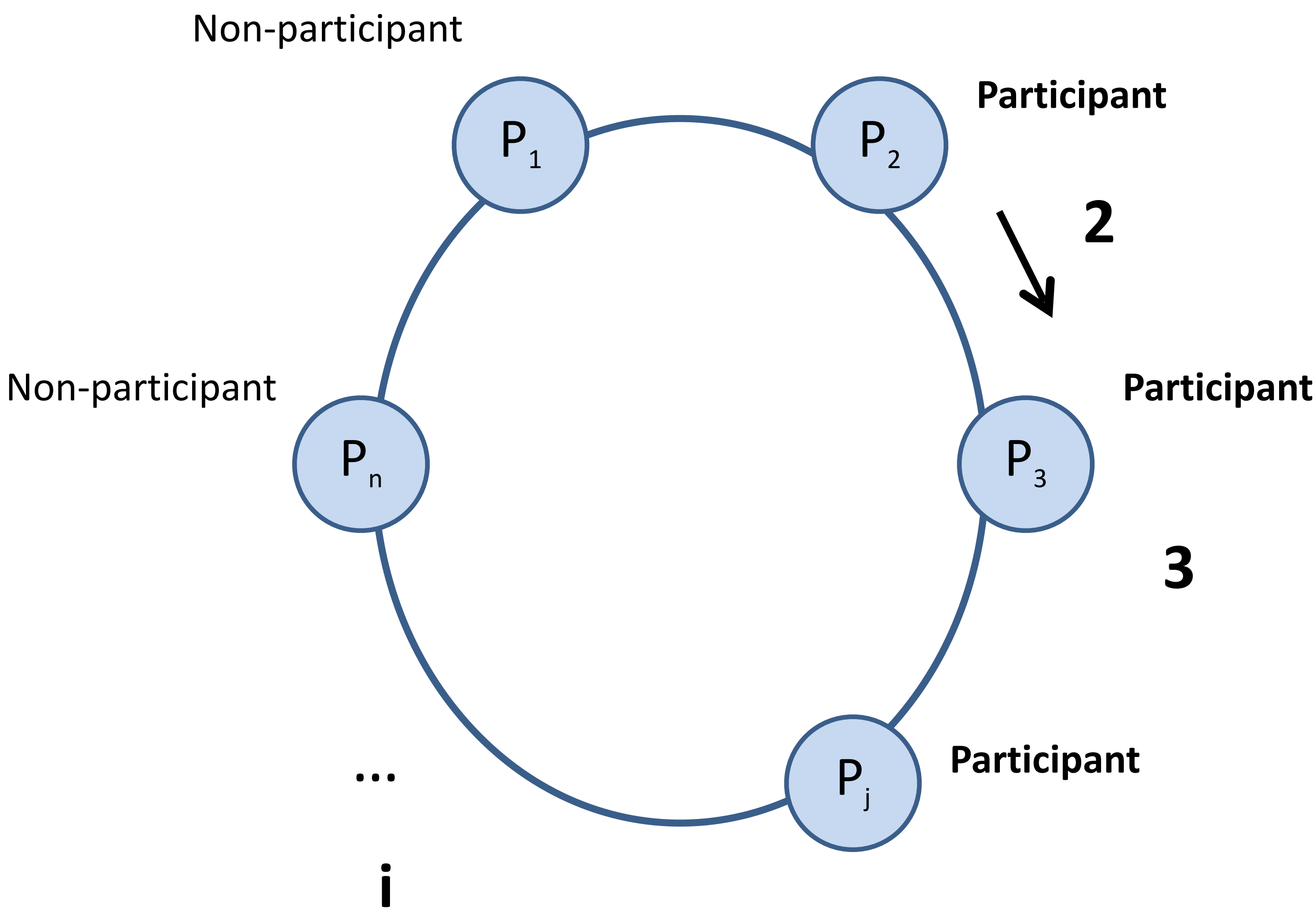
Ring-based algorithm

Calling an election (determine winner)



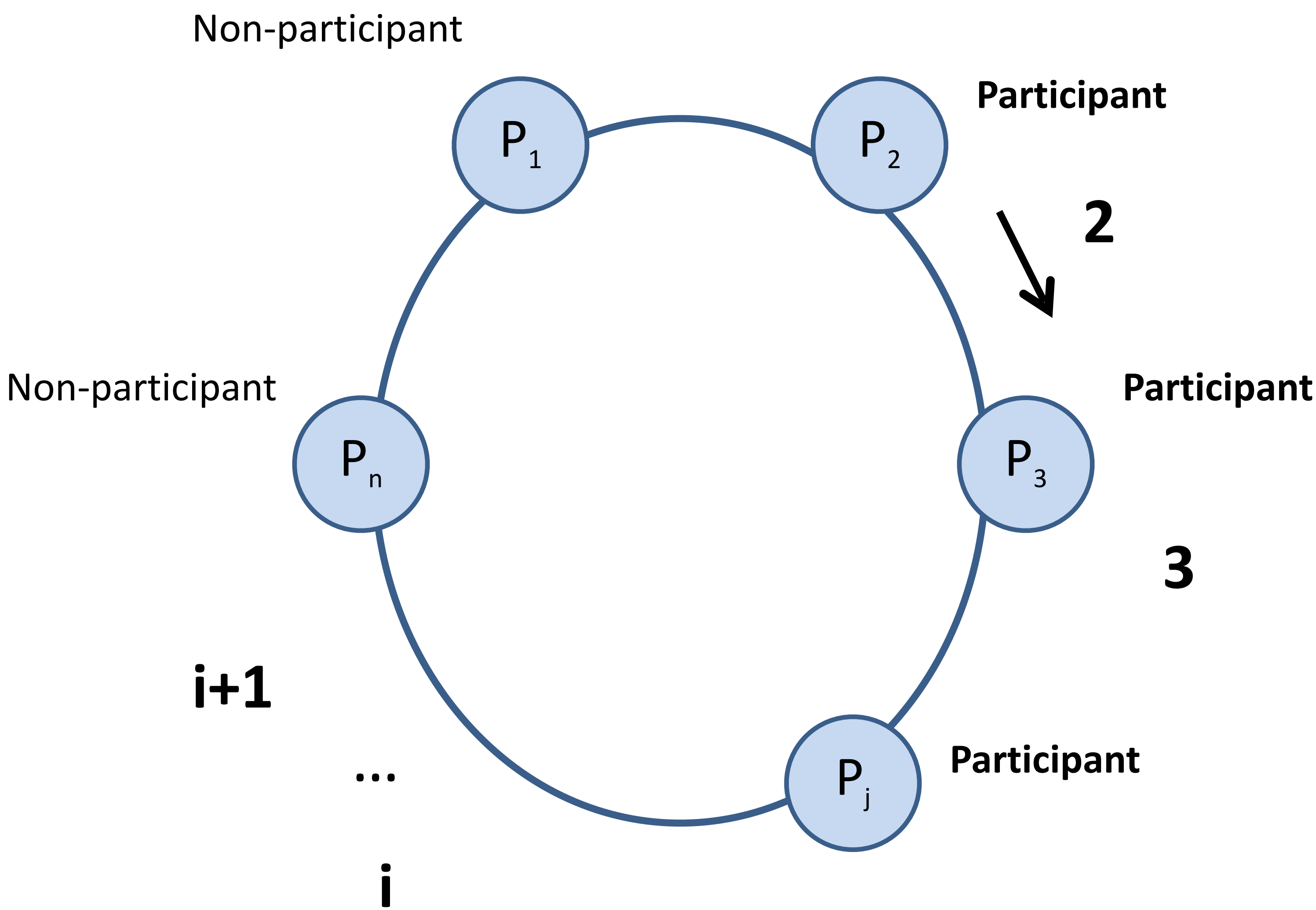
Ring-based algorithm

Calling an election (determine winner)



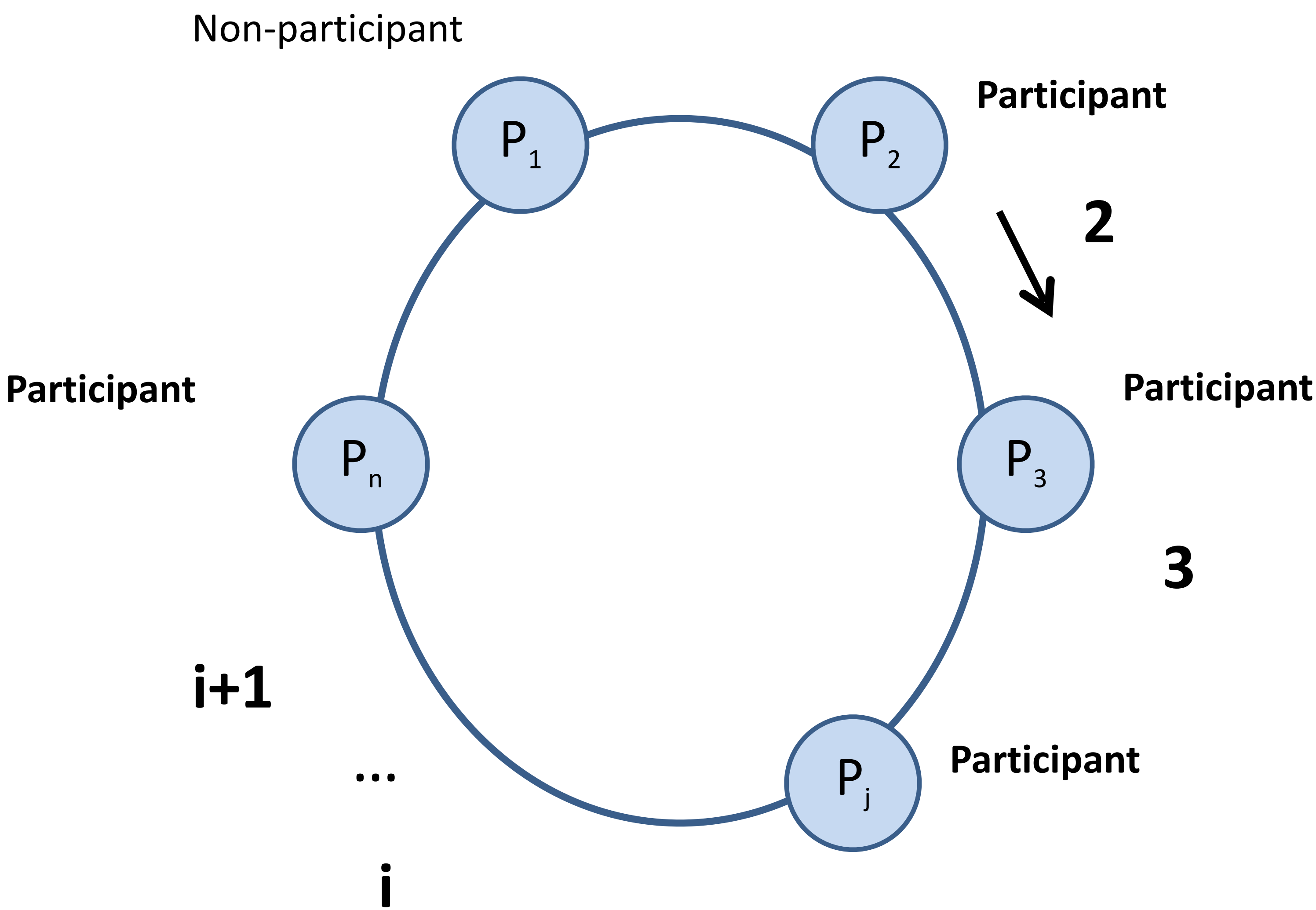
Ring-based algorithm

Calling an election (determine winner)



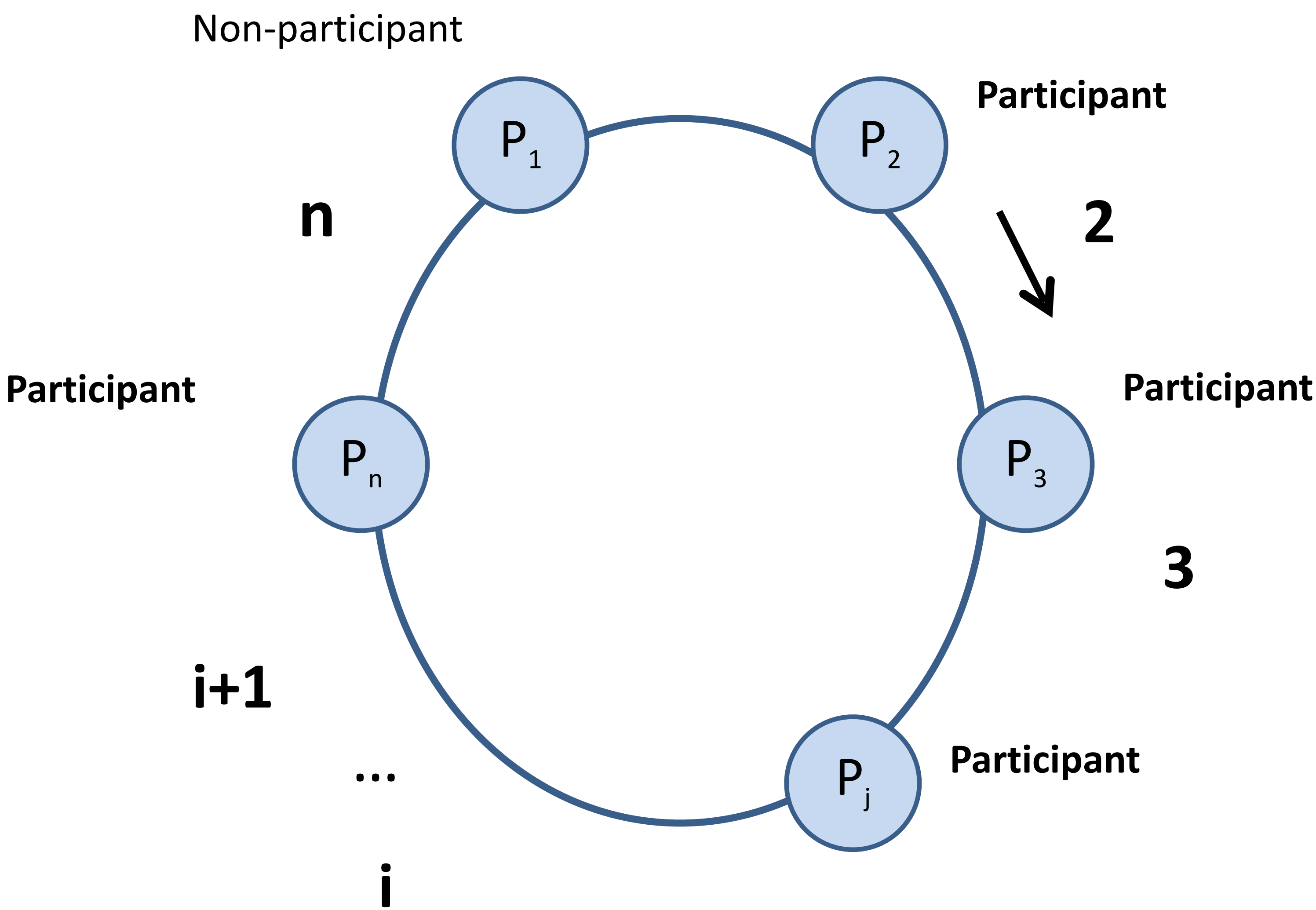
Ring-based algorithm

Calling an election (determine winner)



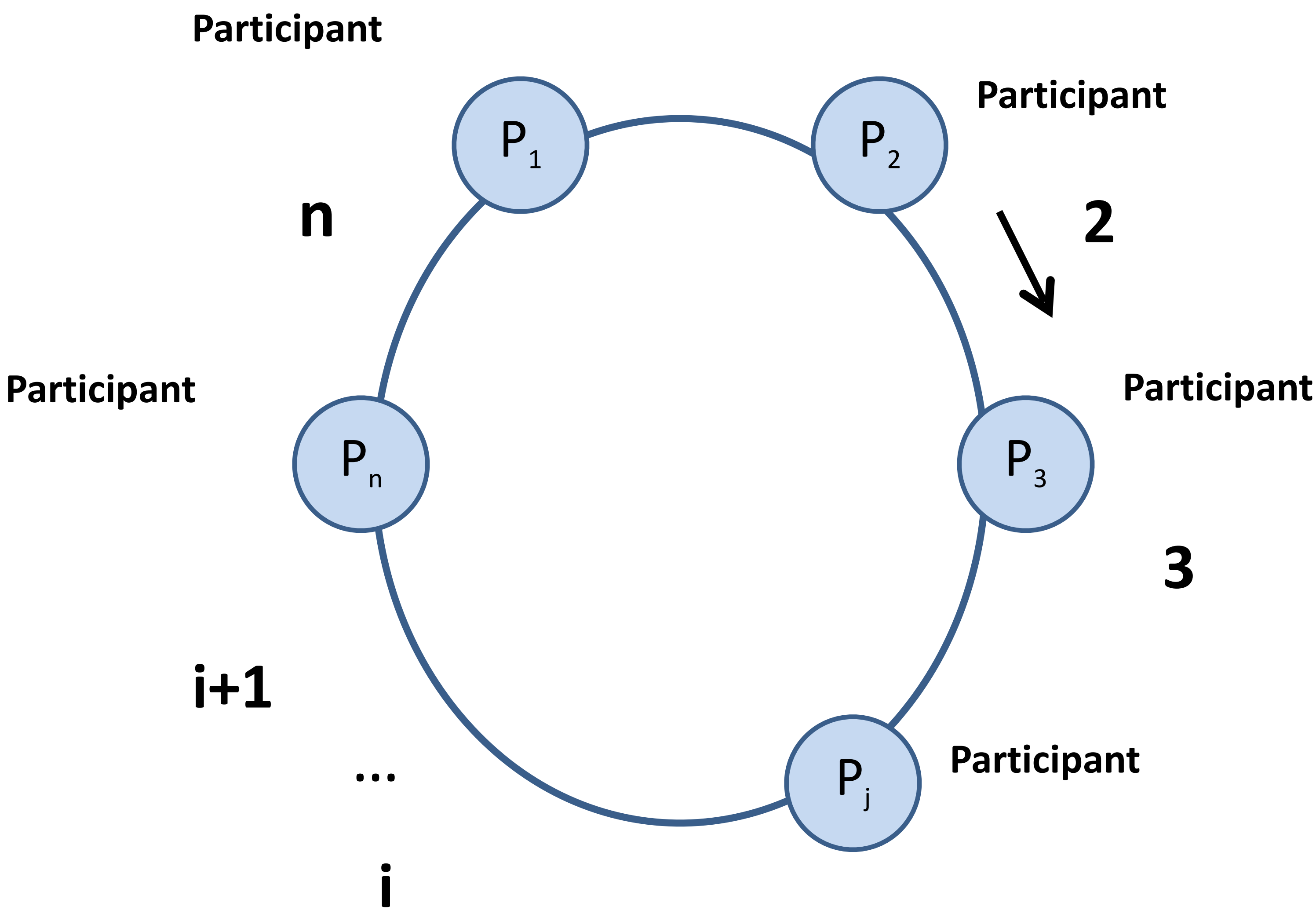
Ring-based algorithm

Calling an election (determine winner)



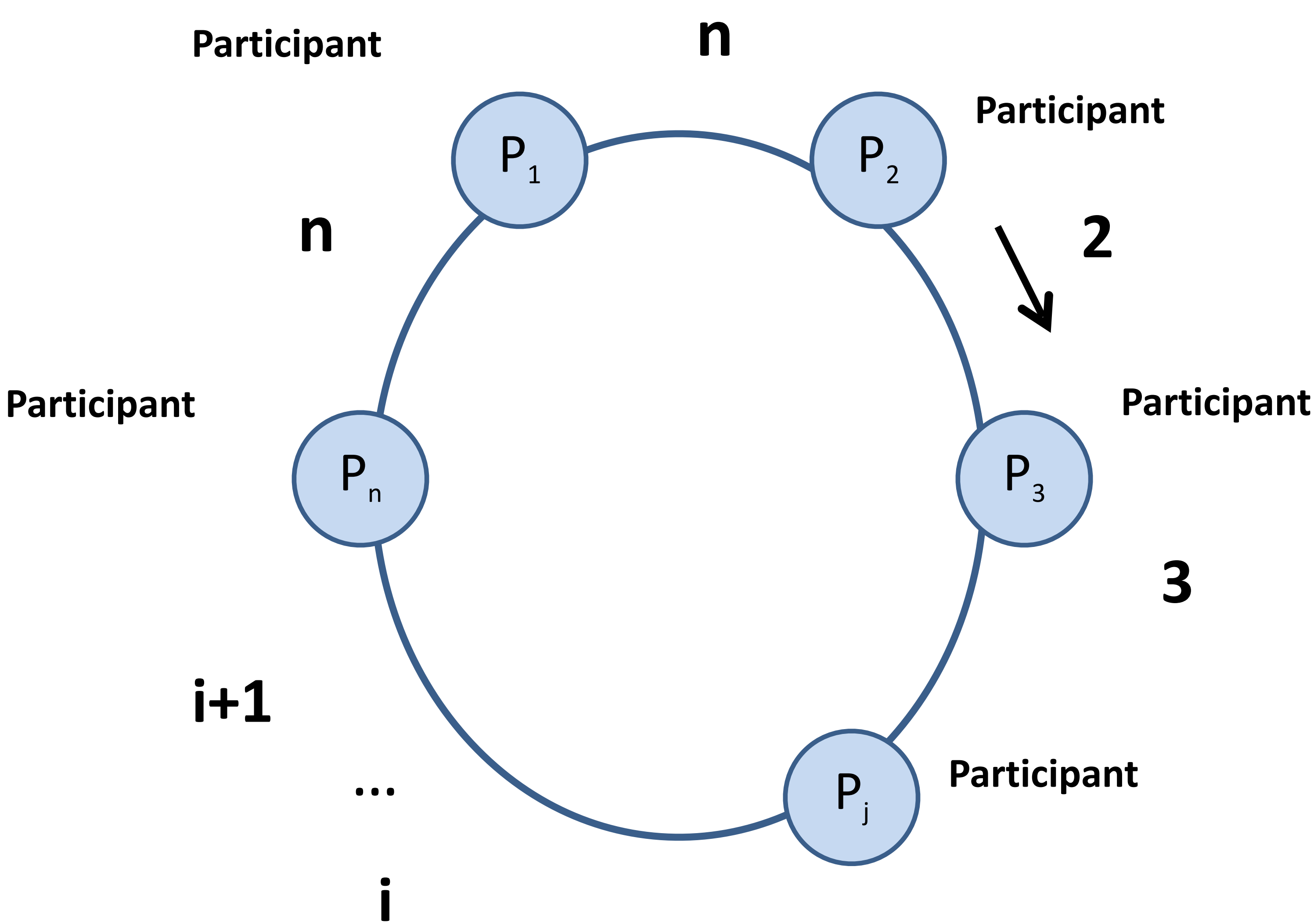
Ring-based algorithm

Calling an election (determine winner)



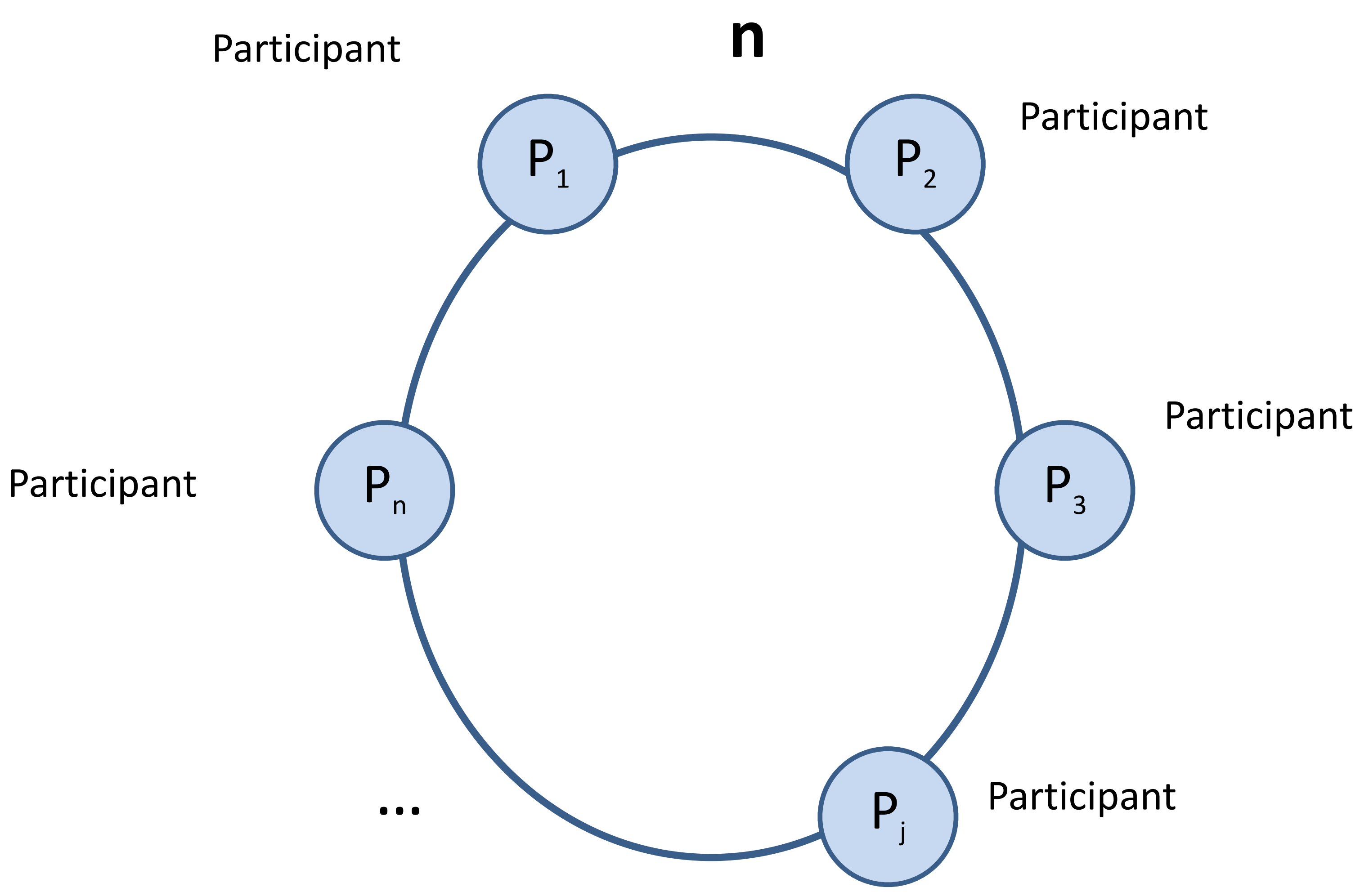
Ring-based algorithm

Calling an election (determine winner)



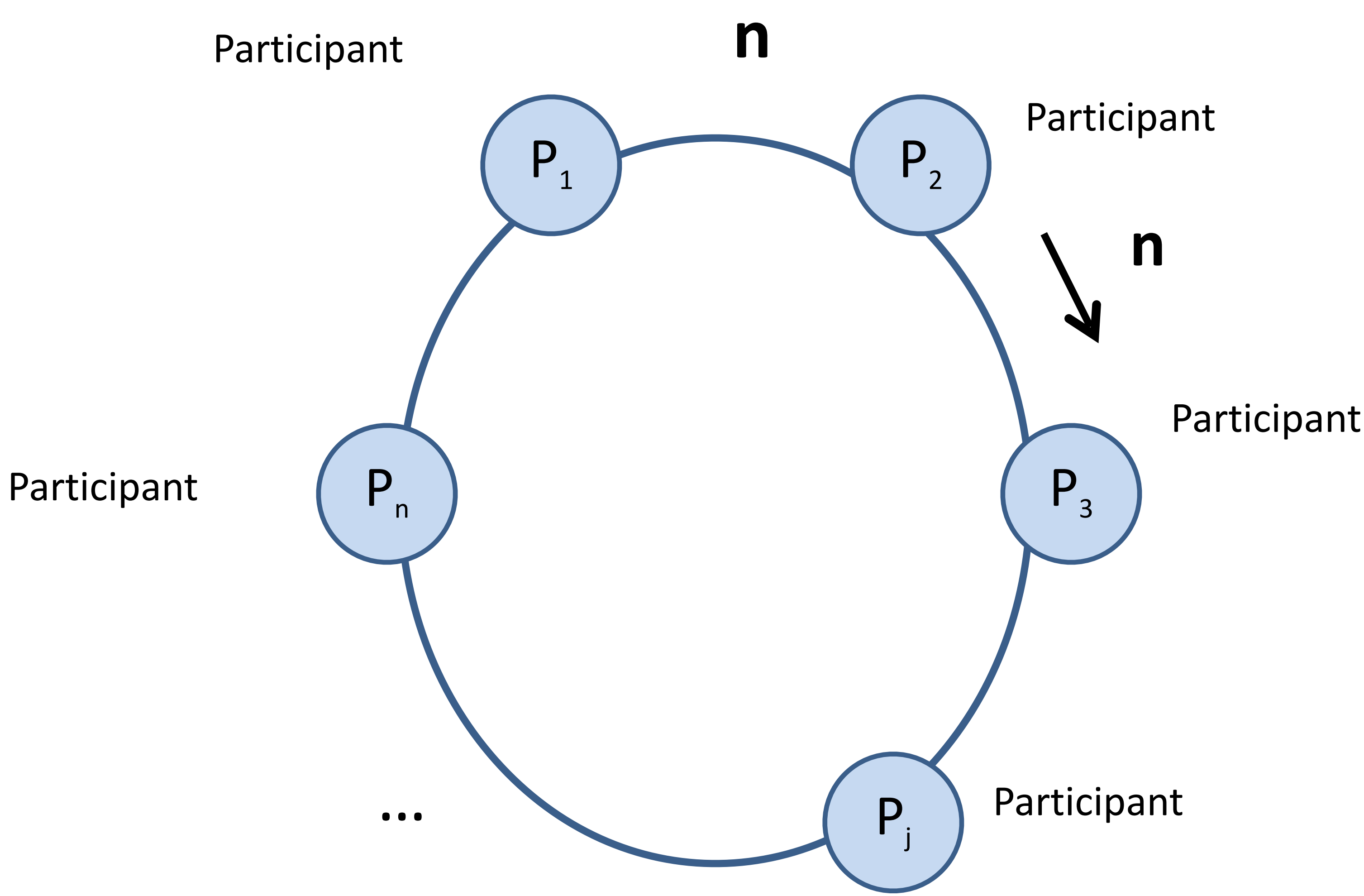
Ring-based algorithm

Calling an election (origin & victory)



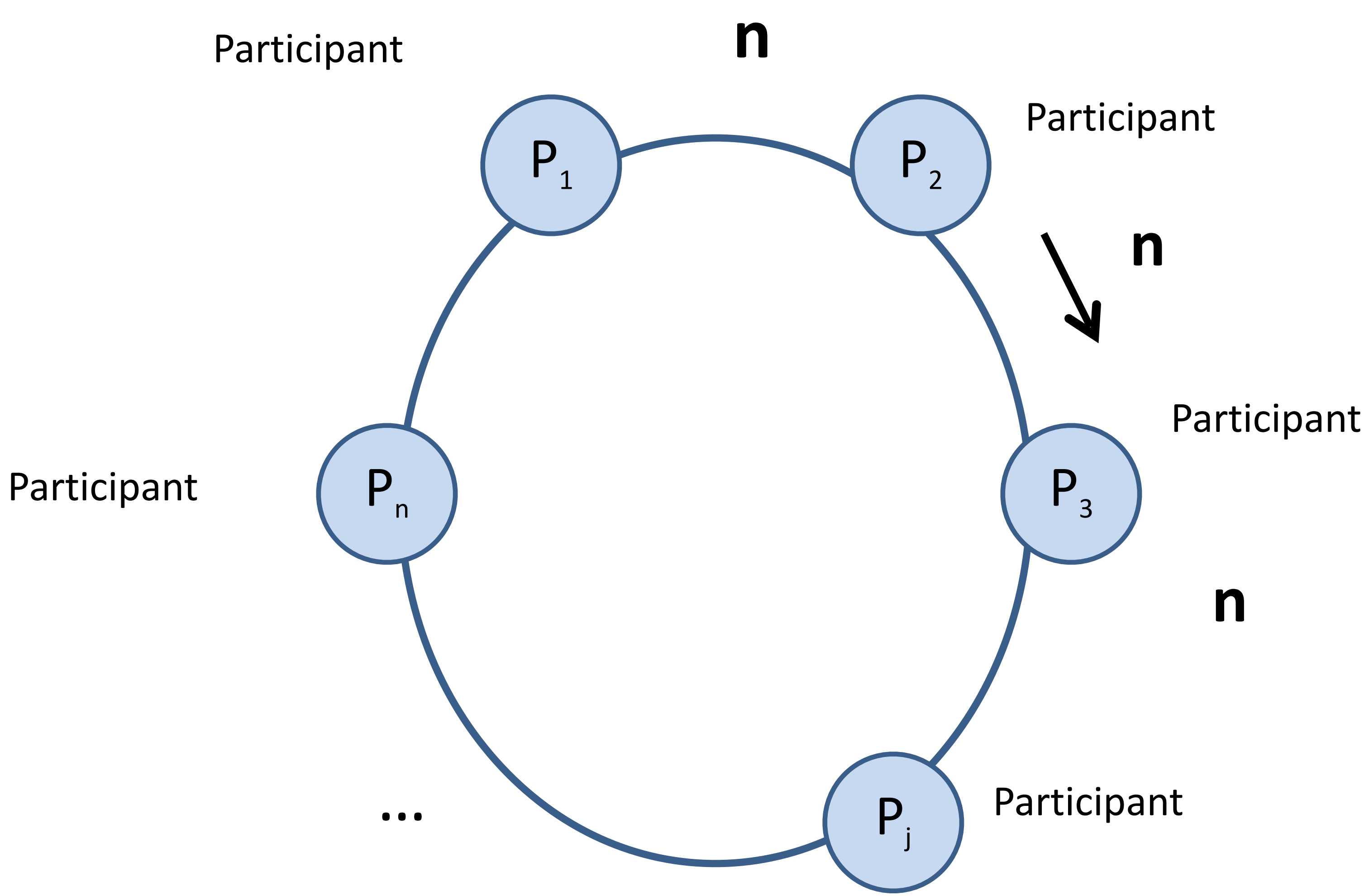
Ring-based algorithm

Calling an election (origin & victory)



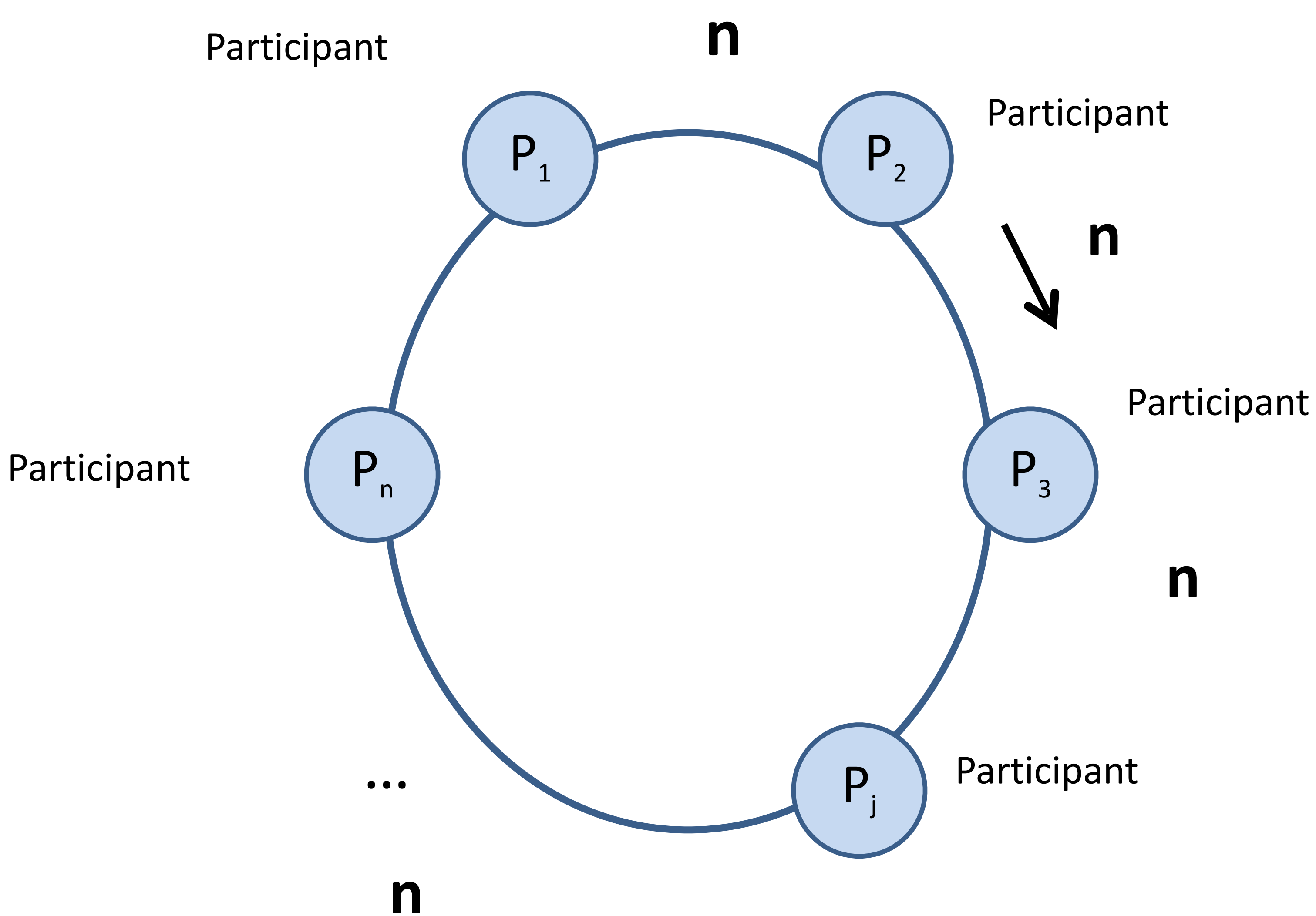
Ring-based algorithm

Calling an election (origin & victory)



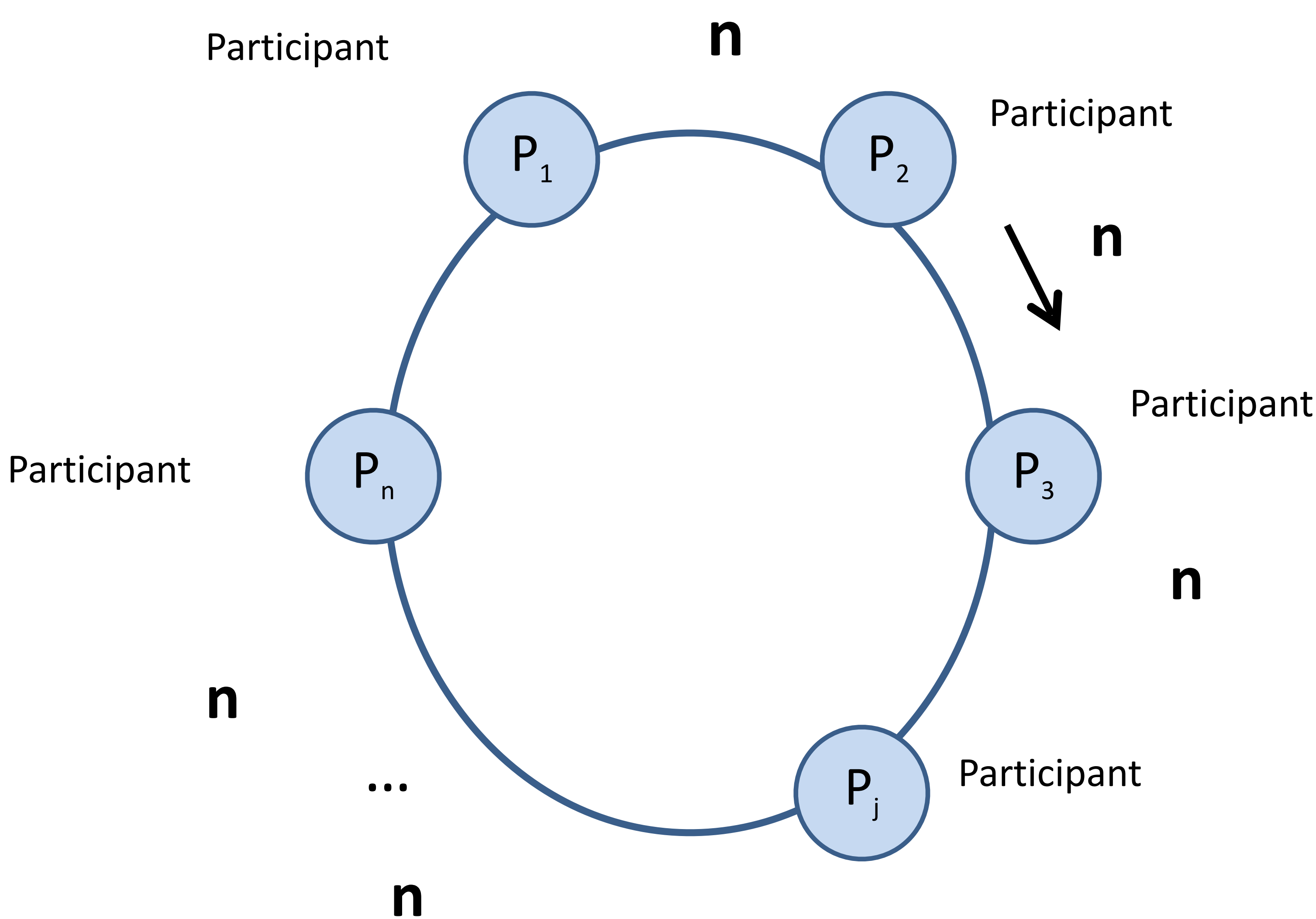
Ring-based algorithm

Calling an election (origin & victory)



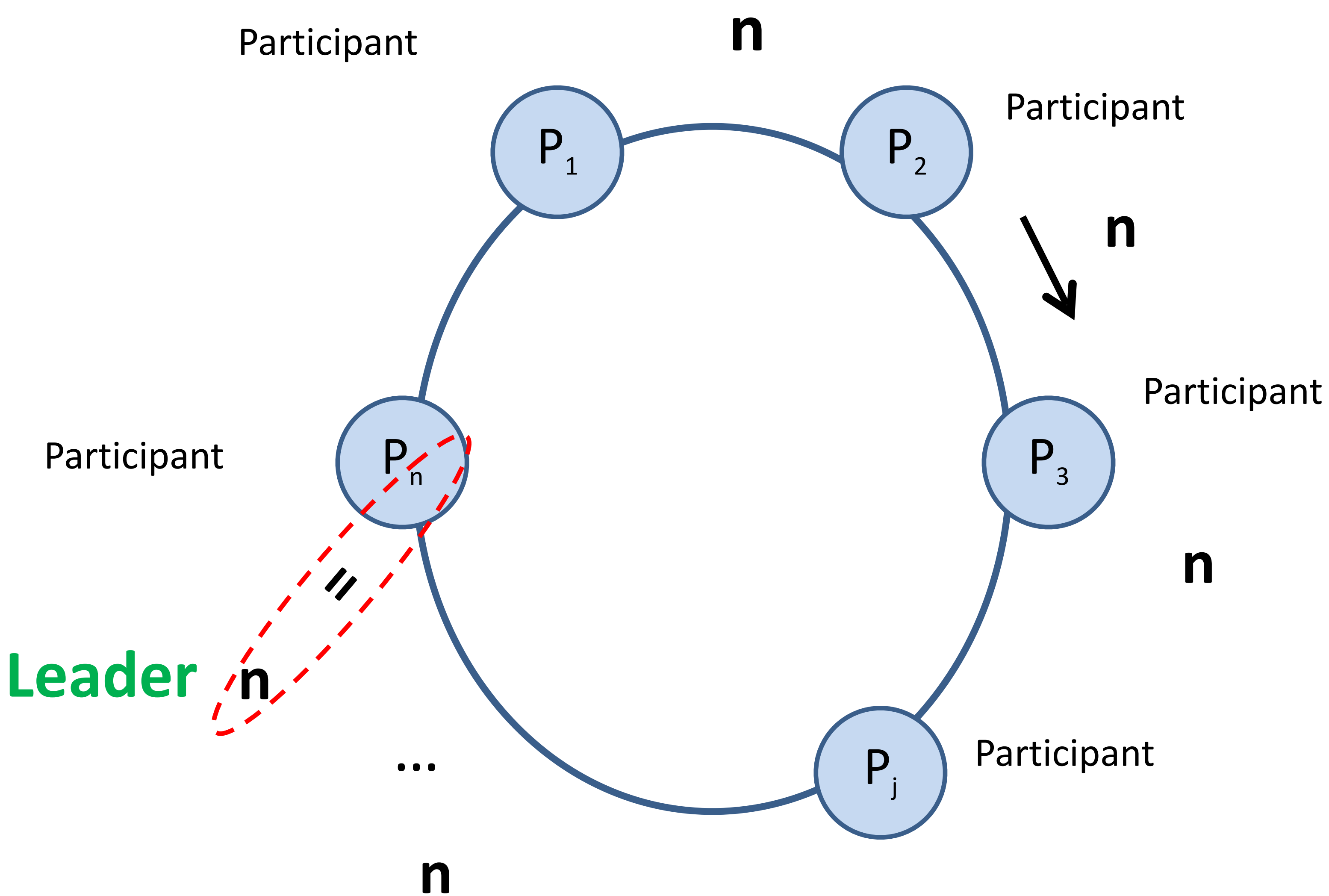
Ring-based algorithm

Calling an election (origin & victory)



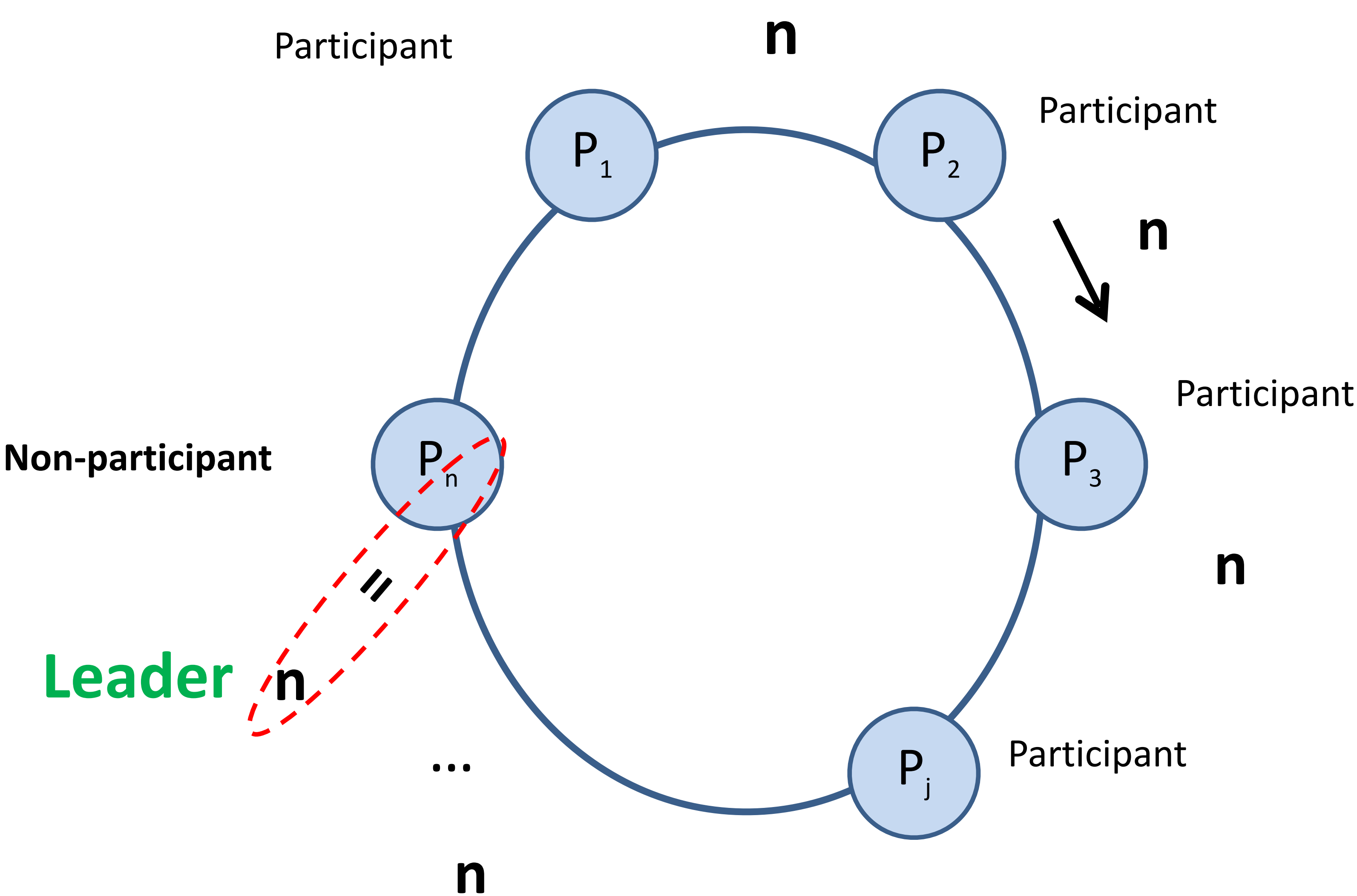
Ring-based algorithm

Calling an election (origin & victory)



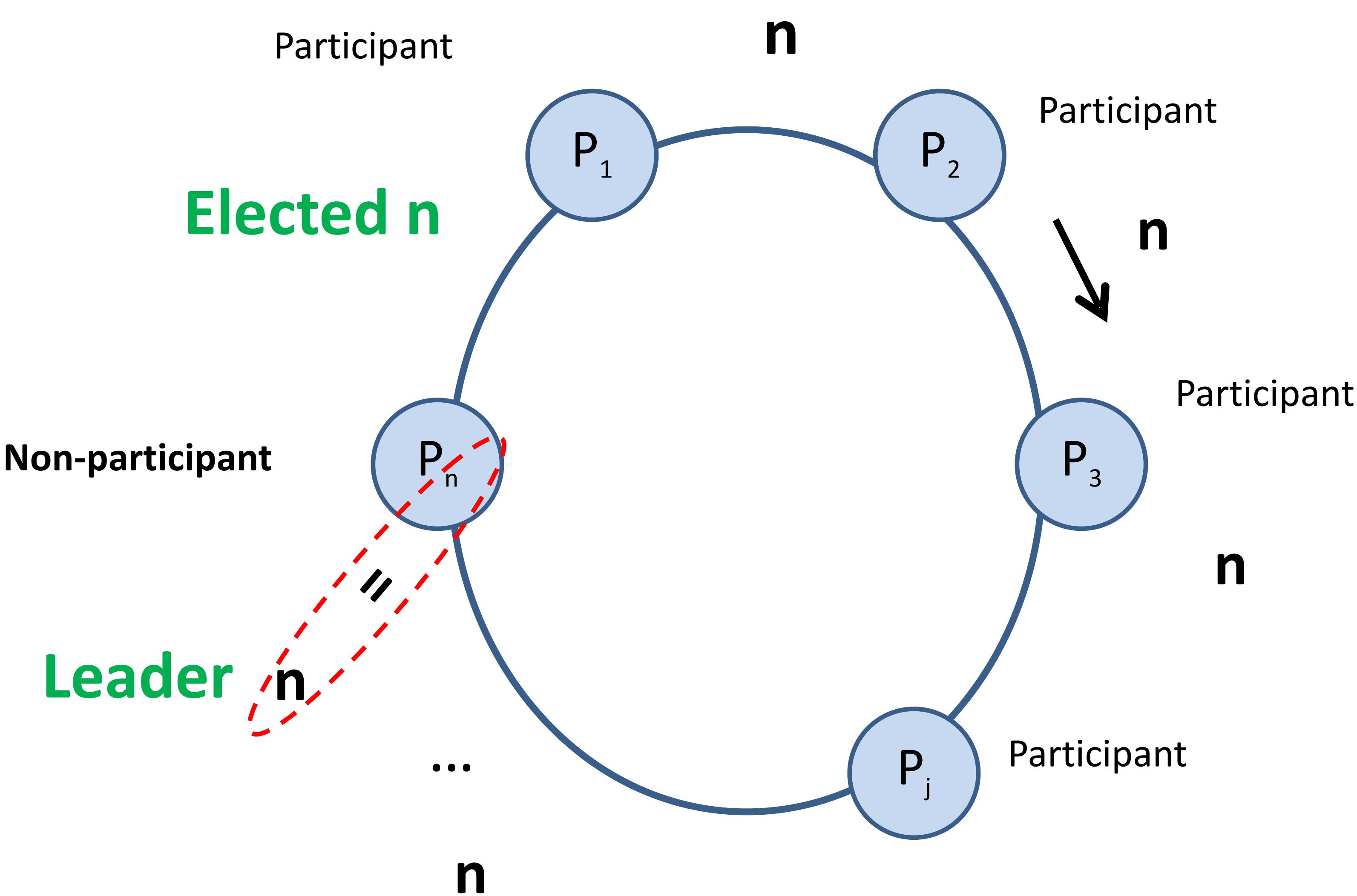
Ring-based algorithm

Calling an election (origin & victory)



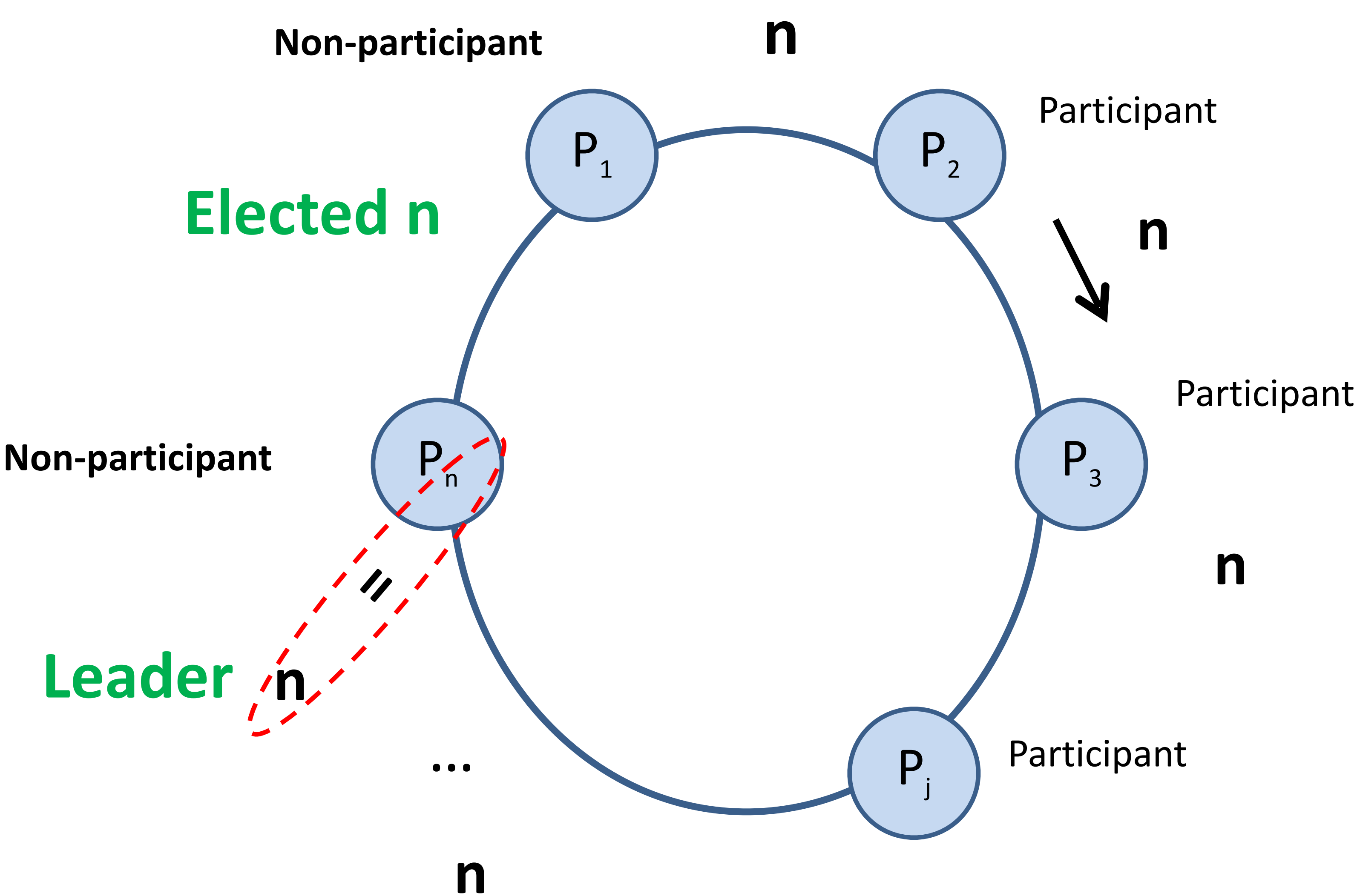
Ring-based algorithm

Calling an election (origin & victory)



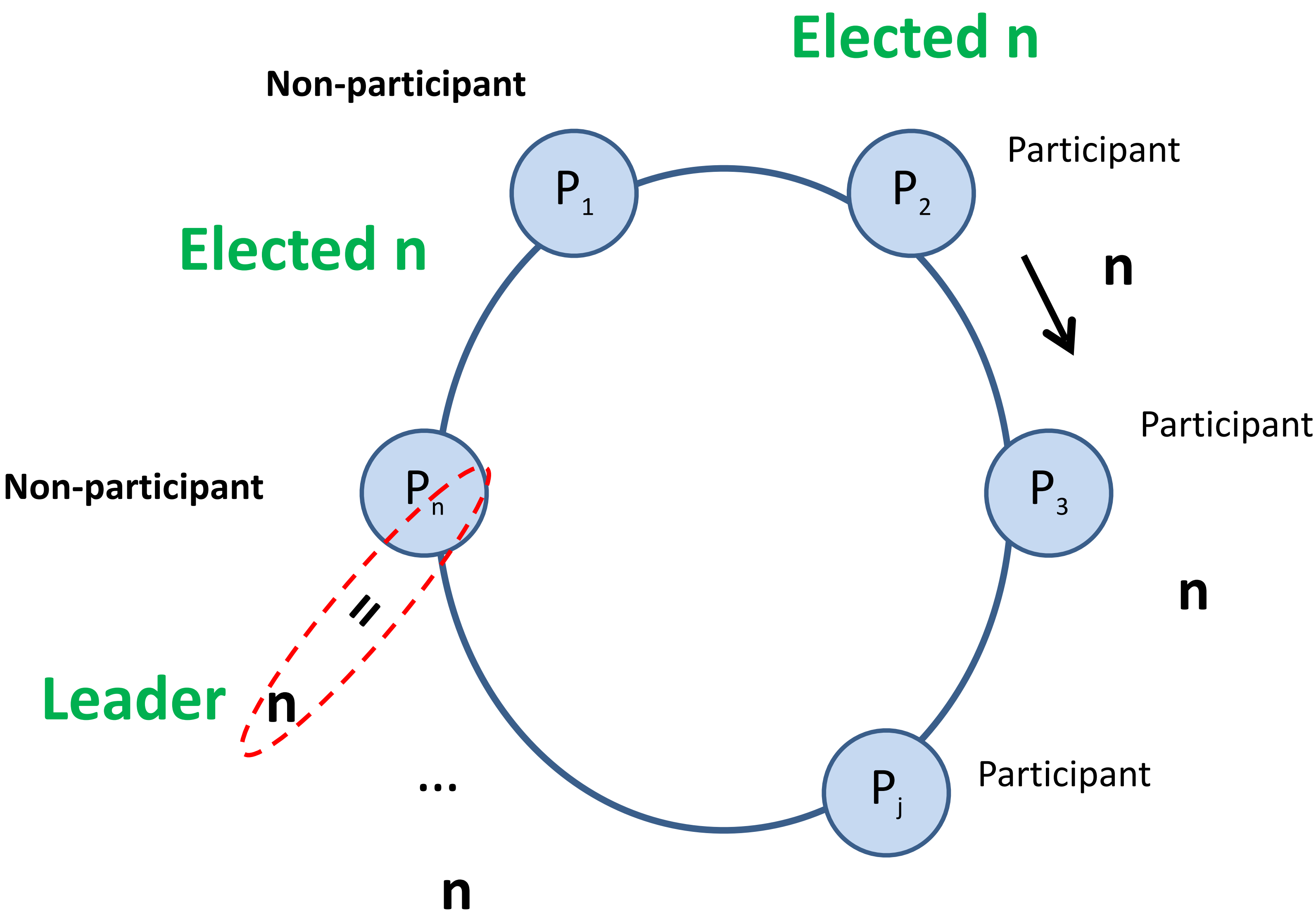
Ring-based algorithm

Calling an election (origin & victory)



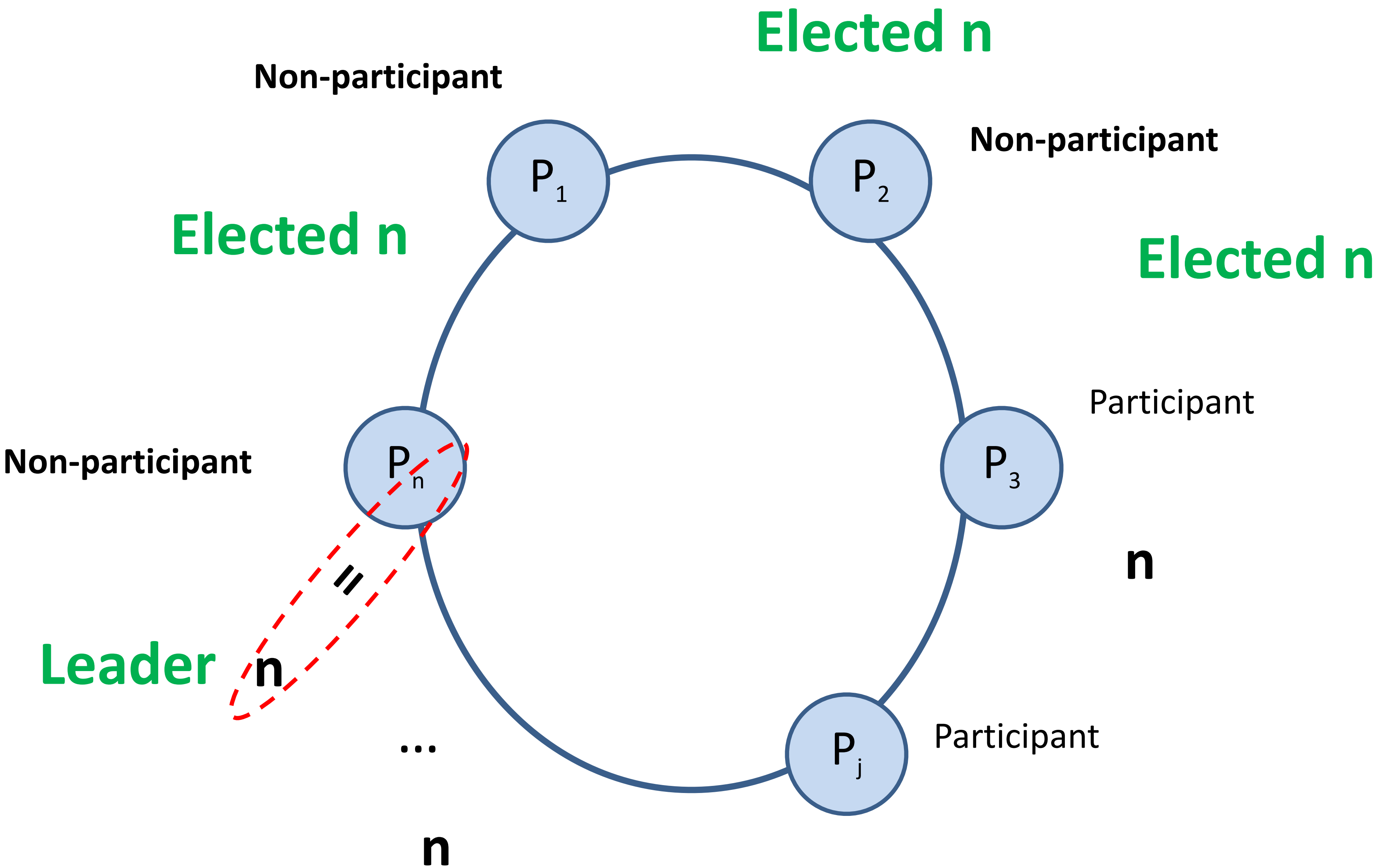
Ring-based algorithm

Calling an election (origin & victory)



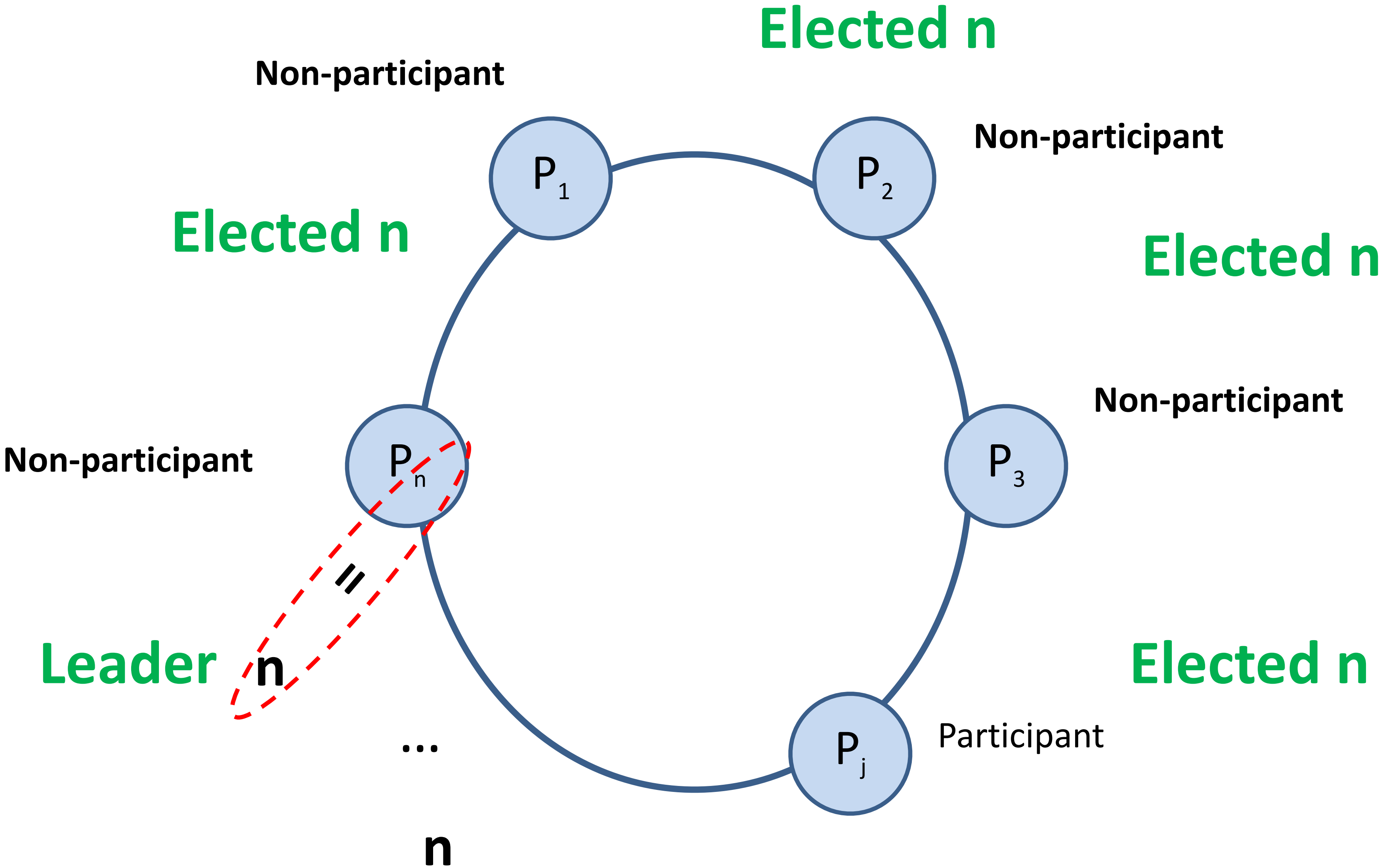
Ring-based algorithm

Calling an election (origin & victory)



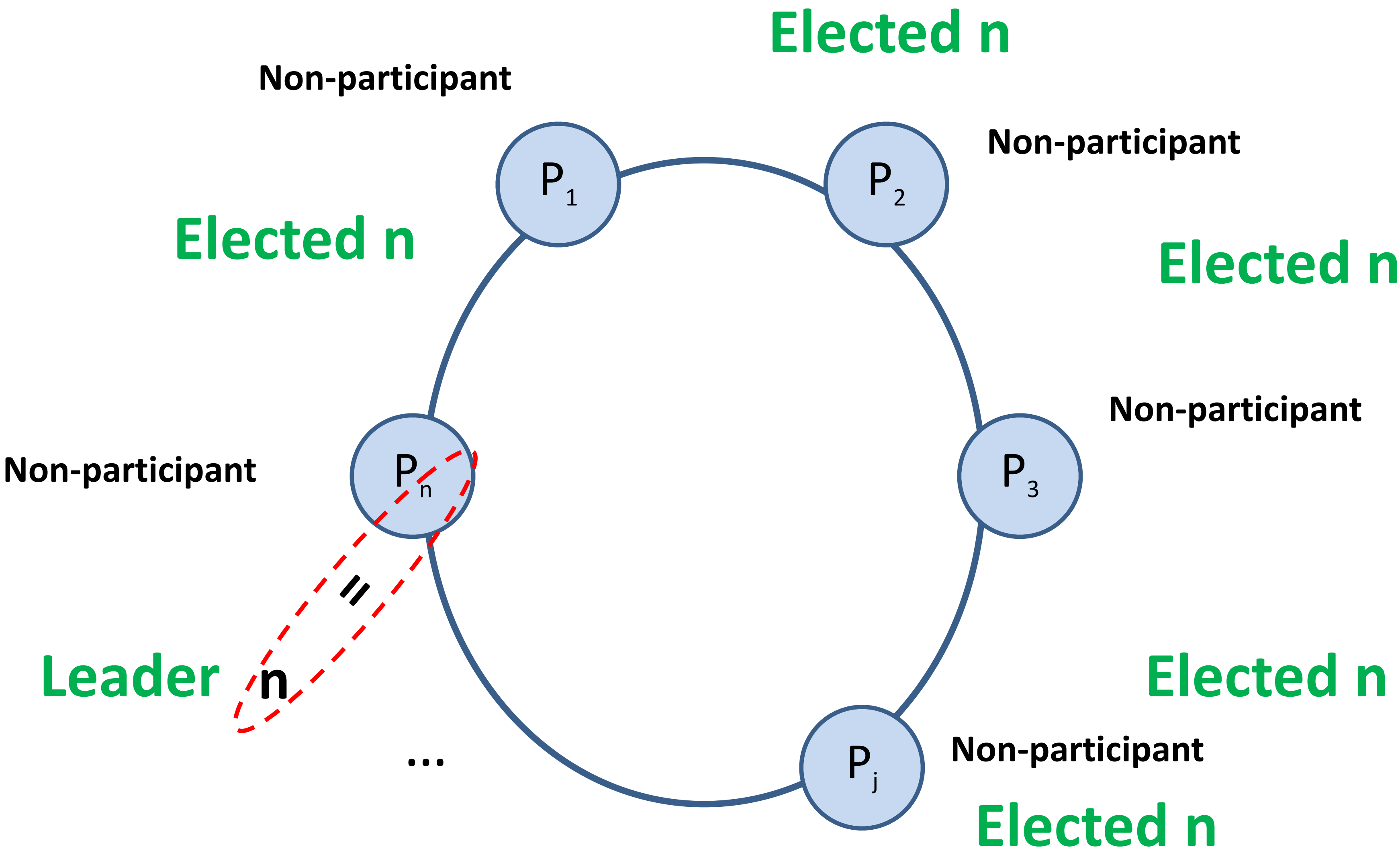
Ring-based algorithm

Calling an election (origin & victory)



Ring-based algorithm

Calling an election (origin & victory)



S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$

Participant

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$

Participant

Forward S_{ID}

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$

Participant

Forward S_{ID}

$S_{ID} > R_{ID}$

Non-participant

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) | **R (receiver)**

$S_{ID} > R_{ID}$ → **Participant**
Forward S_{ID}

$S_{ID} > R_{ID}$ → **Non-participant**
Forward S_{ID} → Participant

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$ → **Participant**
Forward S_{ID}

$S_{ID} > R_{ID}$ → **Non-participant**
Forward S_{ID} → Participant

$S_{ID} < R_{ID}$ → **Non-participant**

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$

Participant

Forward S_{ID}

$S_{ID} > R_{ID}$

Non-participant

Forward S_{ID} → Participant

$S_{ID} < R_{ID}$

Non-participant

Forward R_{ID} → Participant

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$ **Participant**

Forward S_{ID}

$S_{ID} > R_{ID}$ **Non-participant**

Forward S_{ID} → Participant

$S_{ID} < R_{ID}$ **Participant**

$S_{ID} < R_{ID}$ **Non-participant**

Forward R_{ID} → Participant

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) | **R (receiver)**

$S_{ID} > R_{ID}$

Participant
Forward S_{ID}

$S_{ID} > R_{ID}$

Non-participant
Forward S_{ID} → Participant

$S_{ID} < R_{ID}$

Participant
No forwarding (own ID already sent)

$S_{ID} < R_{ID}$

Non-participant
Forward R_{ID} → Participant

S_{ID} ID in message from sender

R_{ID} ID at receiver

Different cases

S (sender) **R (receiver)**

$S_{ID} > R_{ID}$

Participant

Forward S_{ID}

If $\mathbf{S_{ID} = R_{ID}}$, it follows

R elected as leader

$S_{ID} > R_{ID}$

Non-participant

Forward S_{ID} → Participant

$S_{ID} < R_{ID}$

Participant

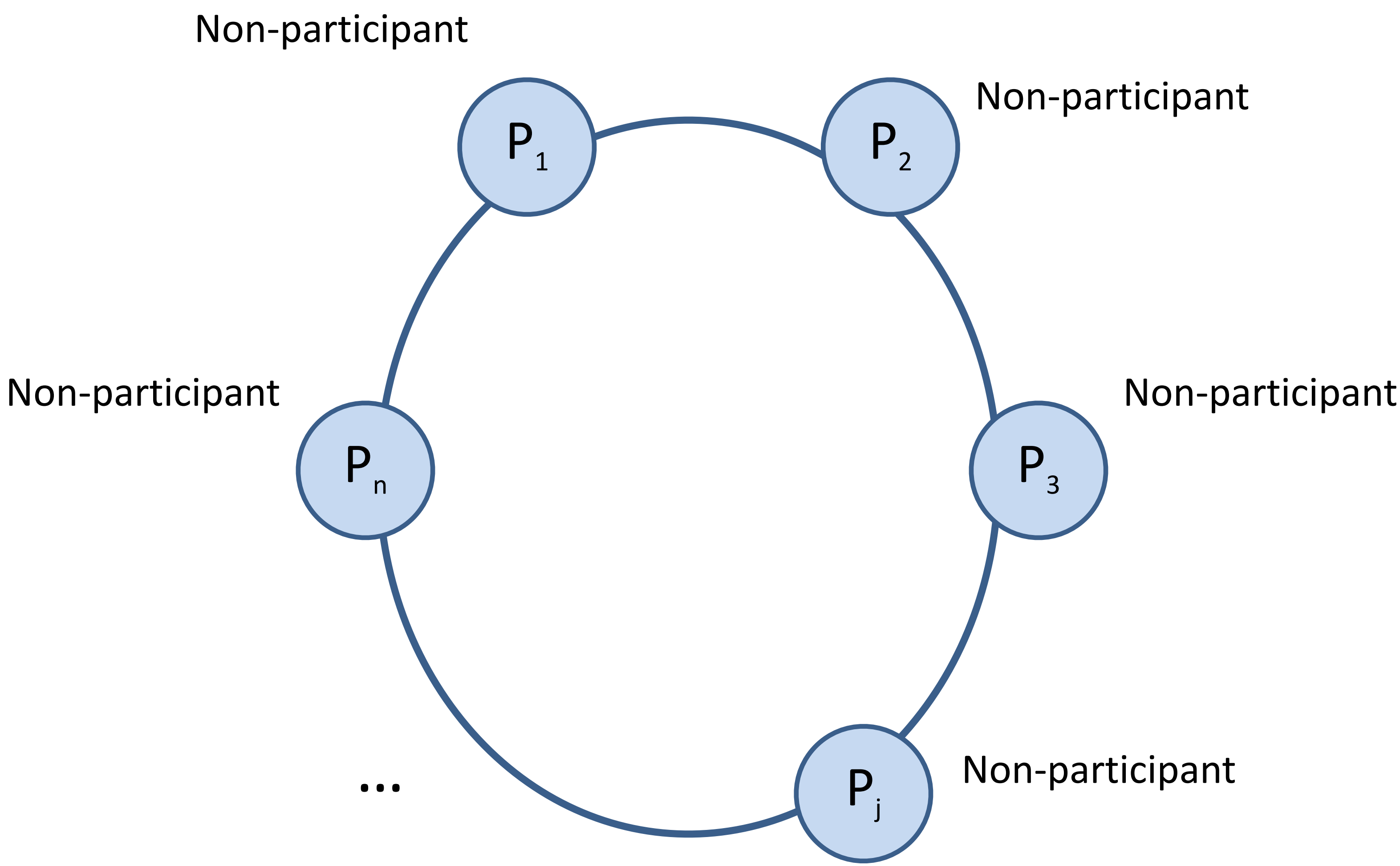
No forwarding (own ID already sent)

$S_{ID} < R_{ID}$

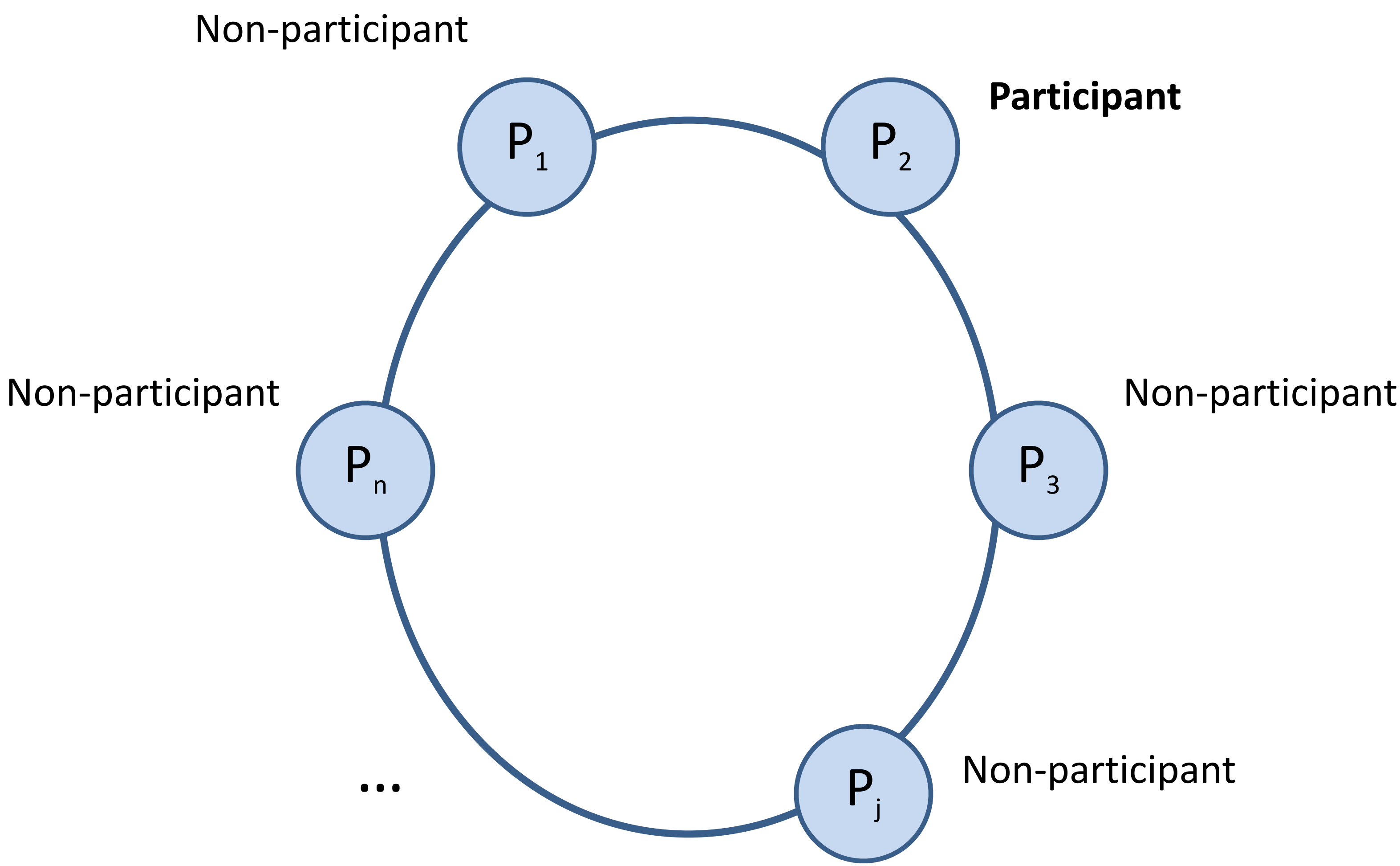
Non-participant

Forward R_{ID} → Participant

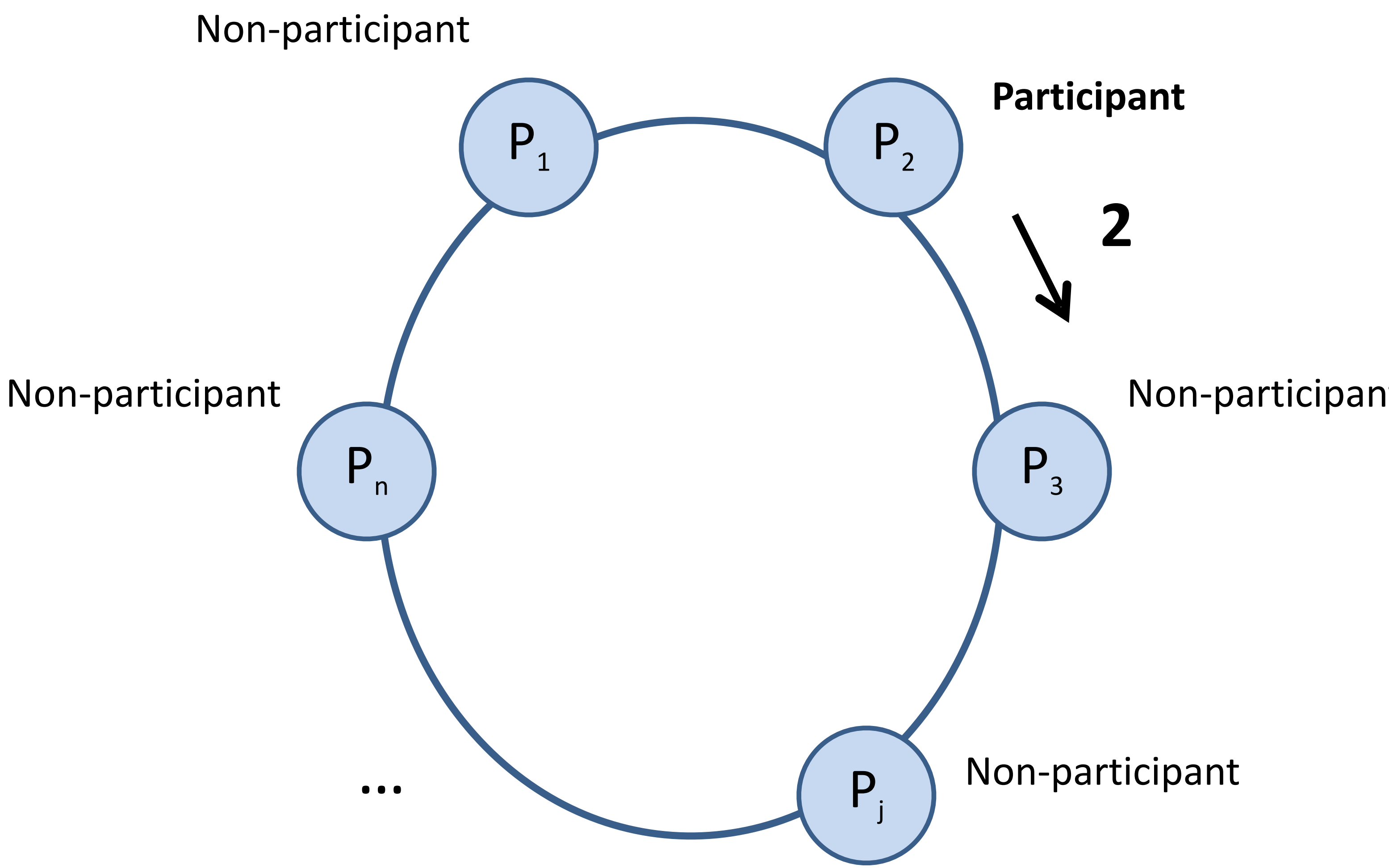
Ring-based algorithm:
Concurrent election start



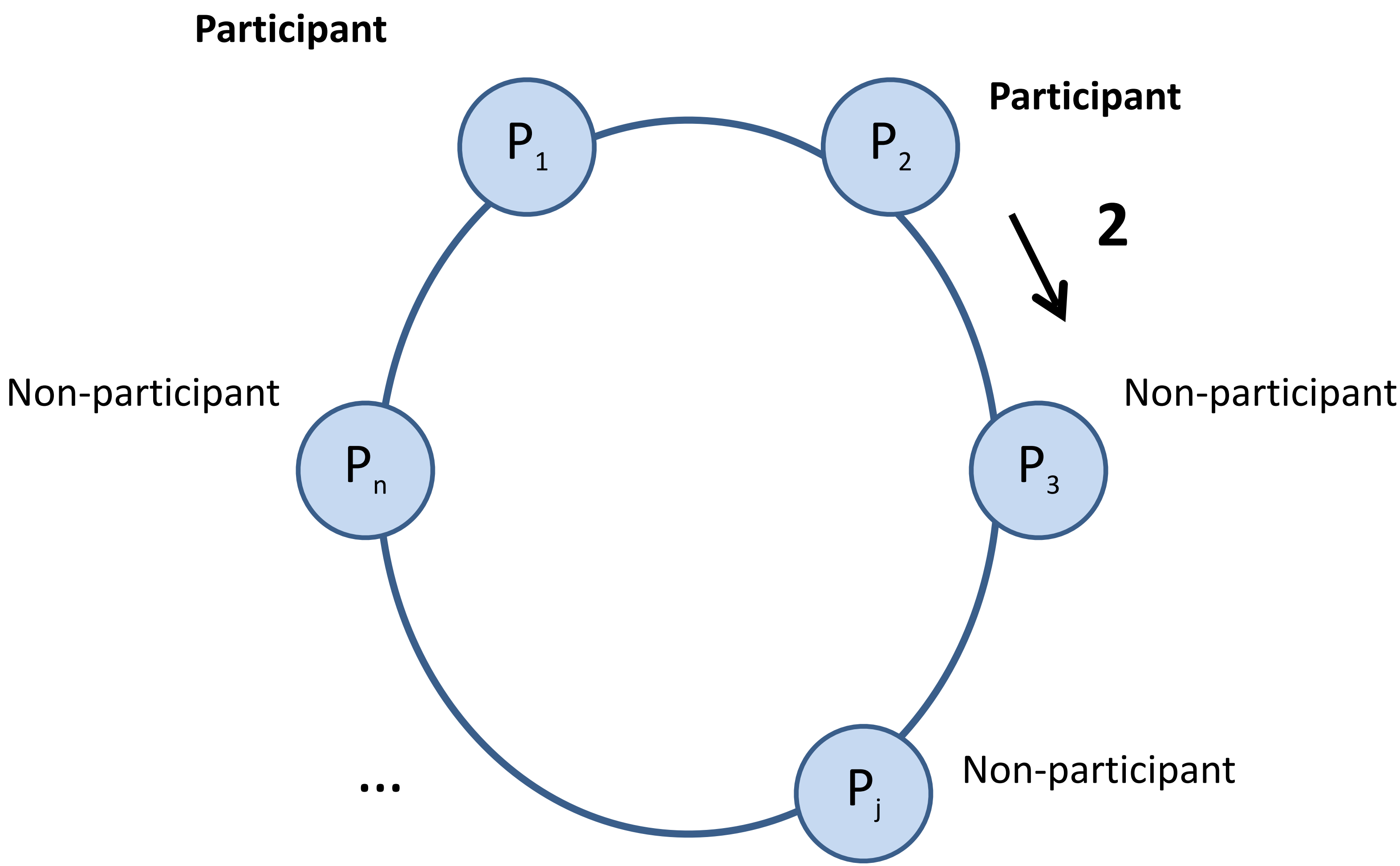
Ring-based algorithm:
Concurrent election start



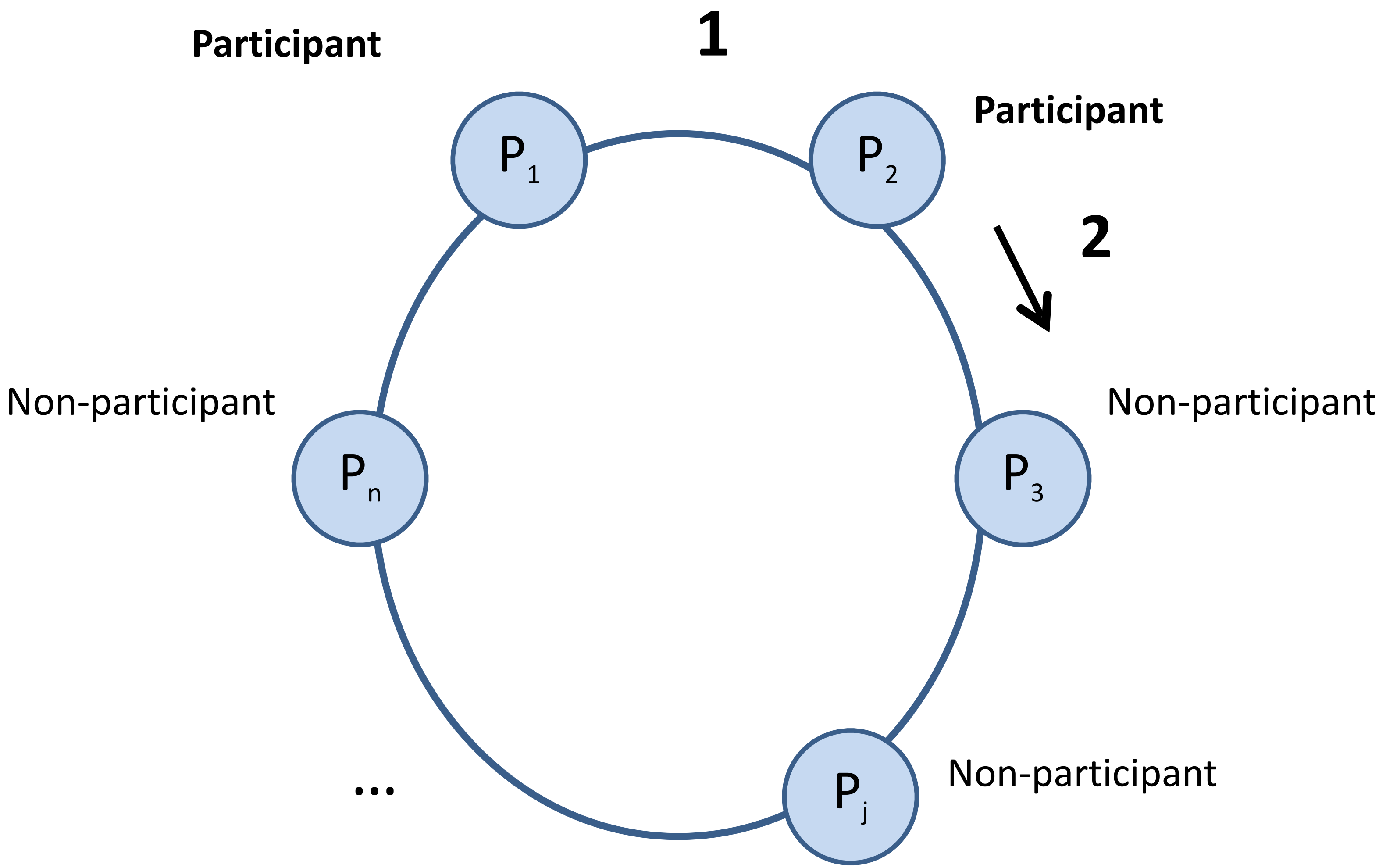
Ring-based algorithm:
Concurrent election start



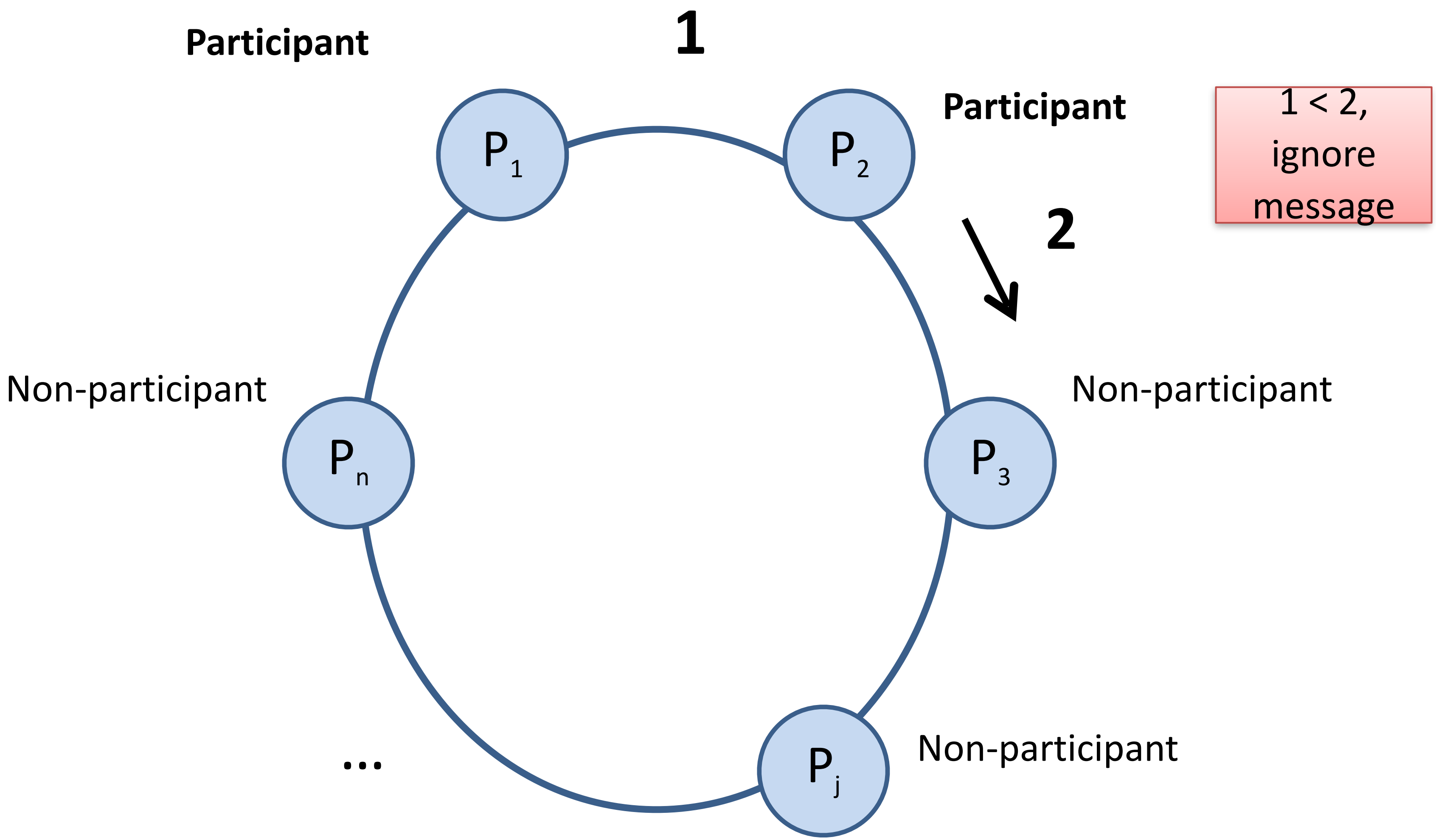
Ring-based algorithm:
Concurrent election start



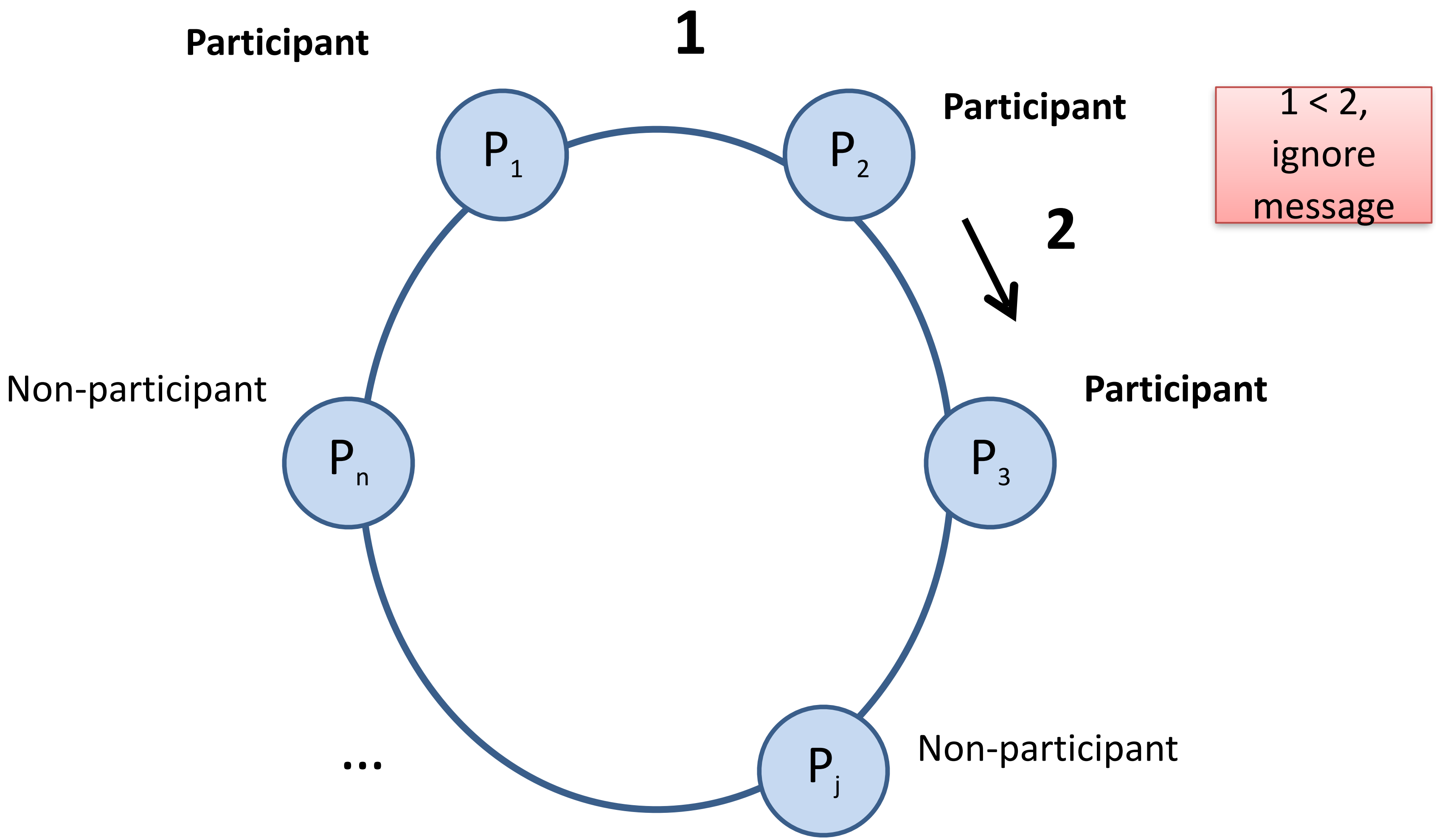
Ring-based algorithm:
Concurrent election start



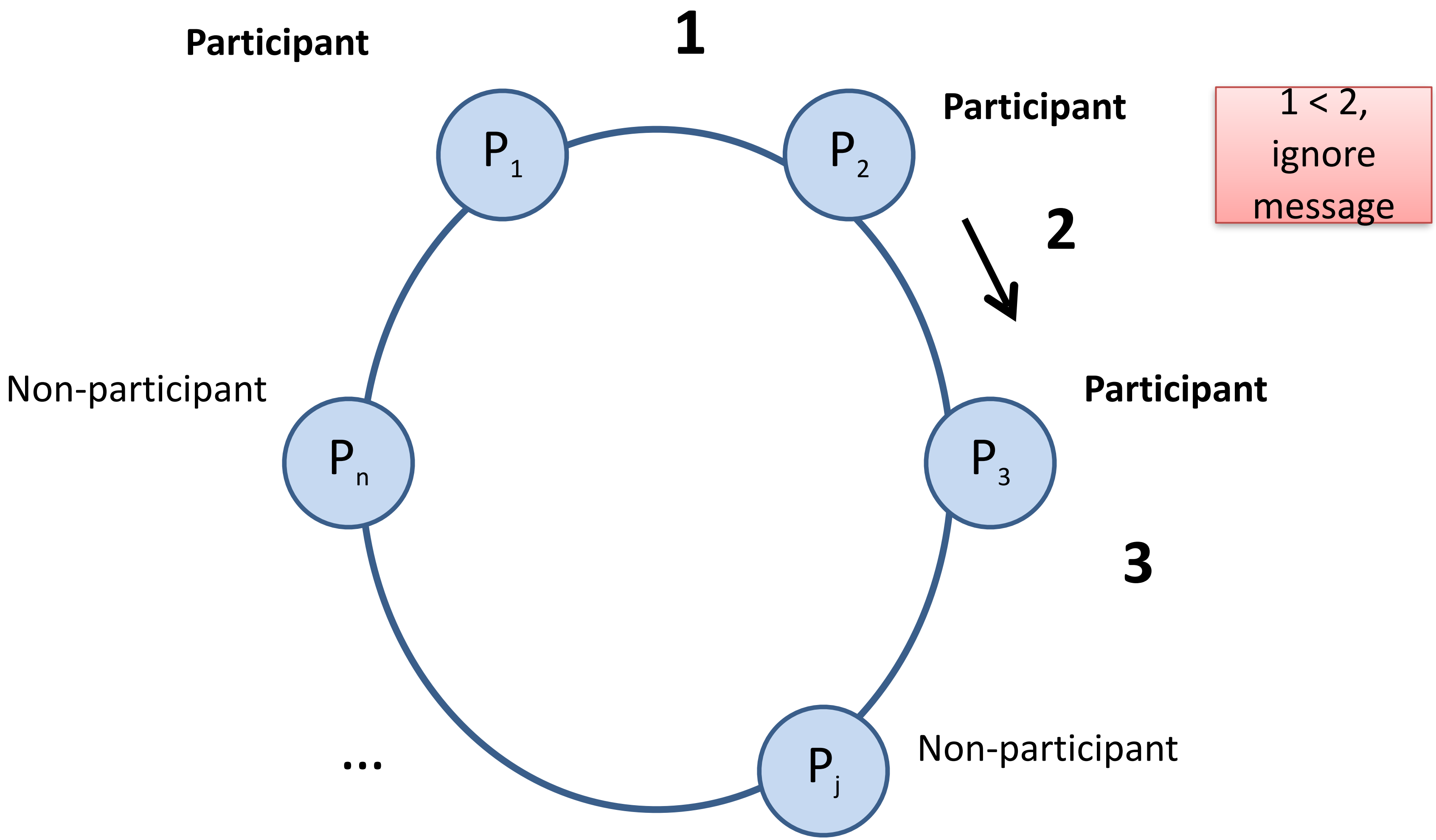
Ring-based algorithm:
Concurrent election start



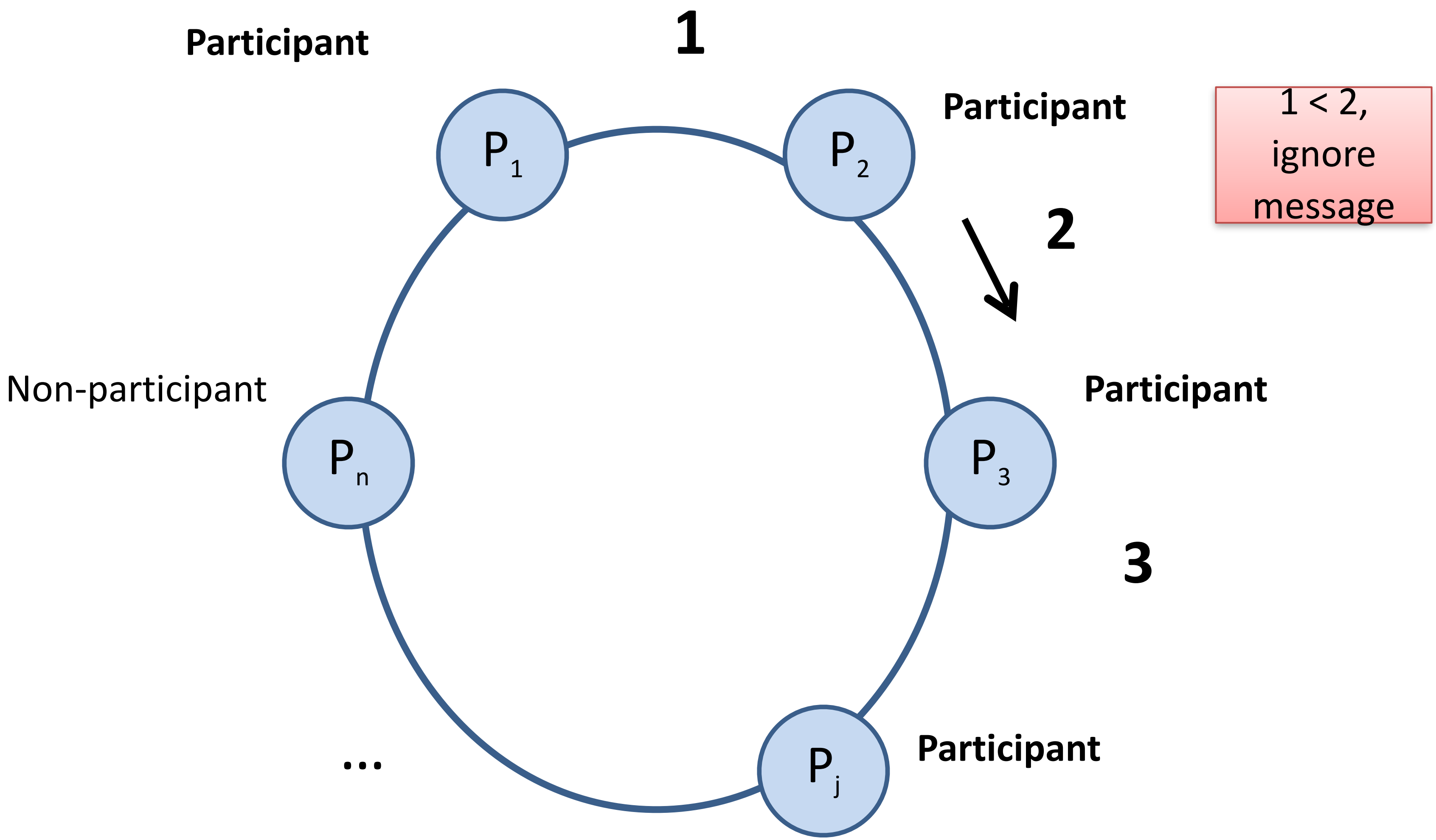
Ring-based algorithm:
Concurrent election start



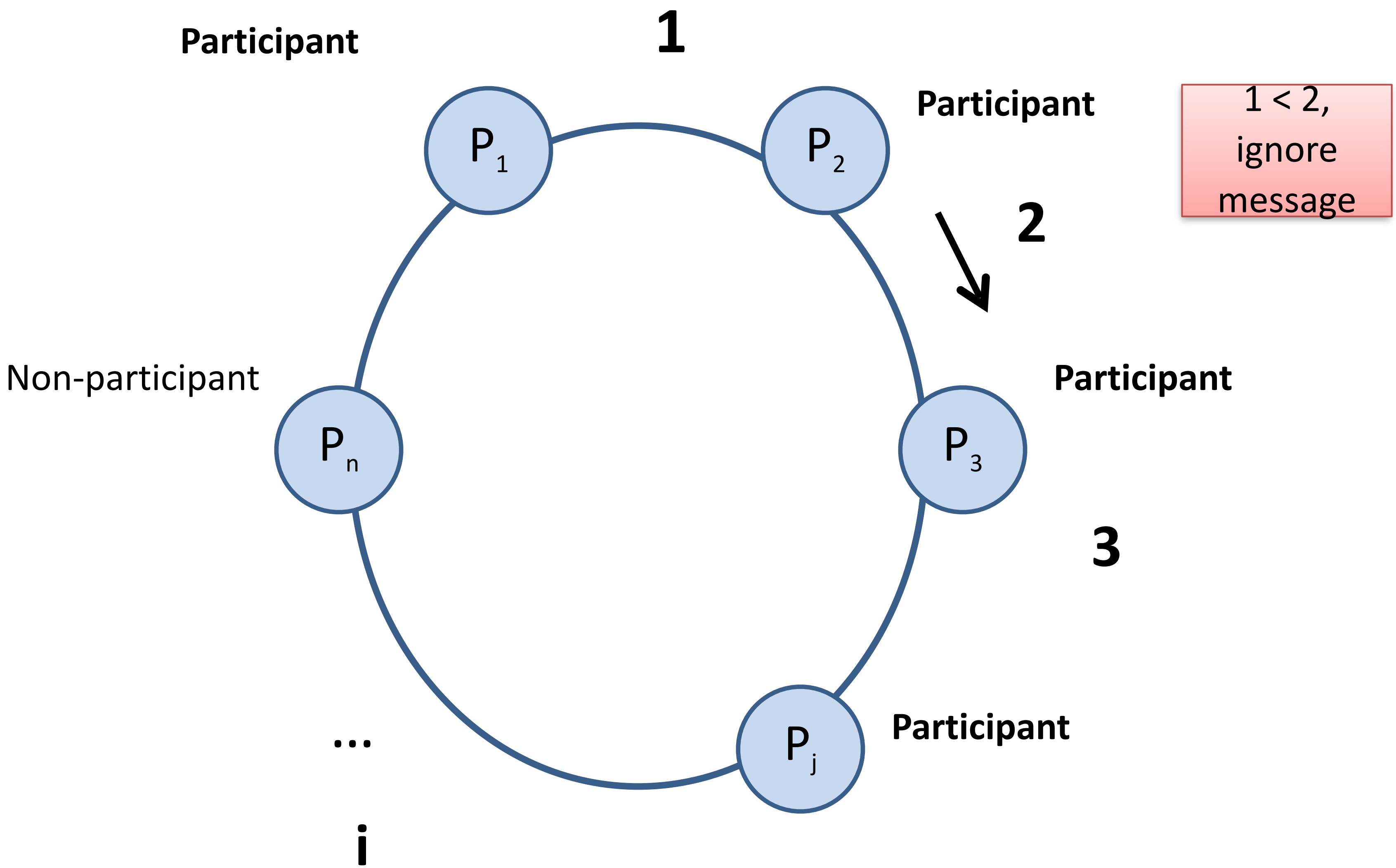
Ring-based algorithm:
Concurrent election start



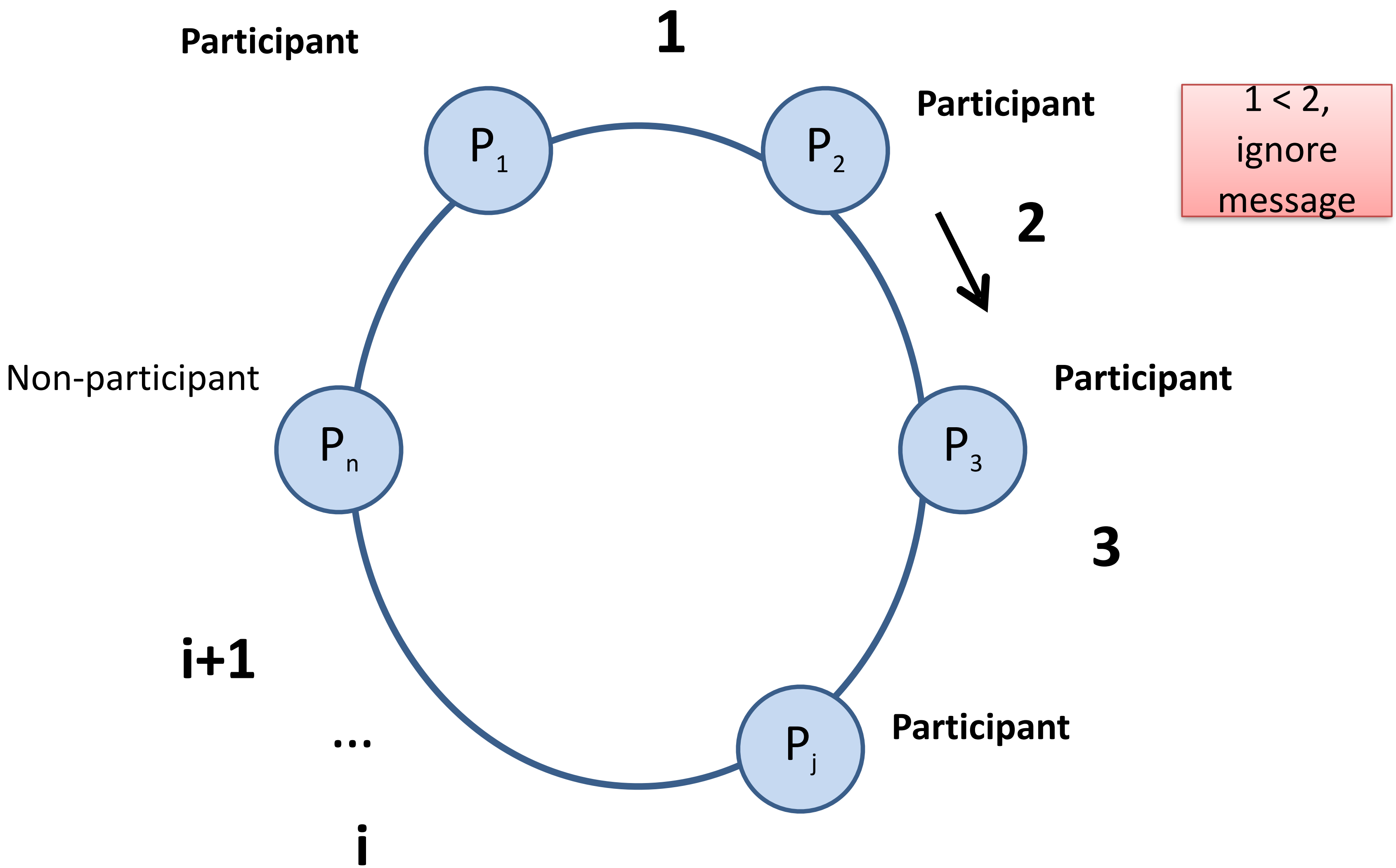
Ring-based algorithm:
Concurrent election start



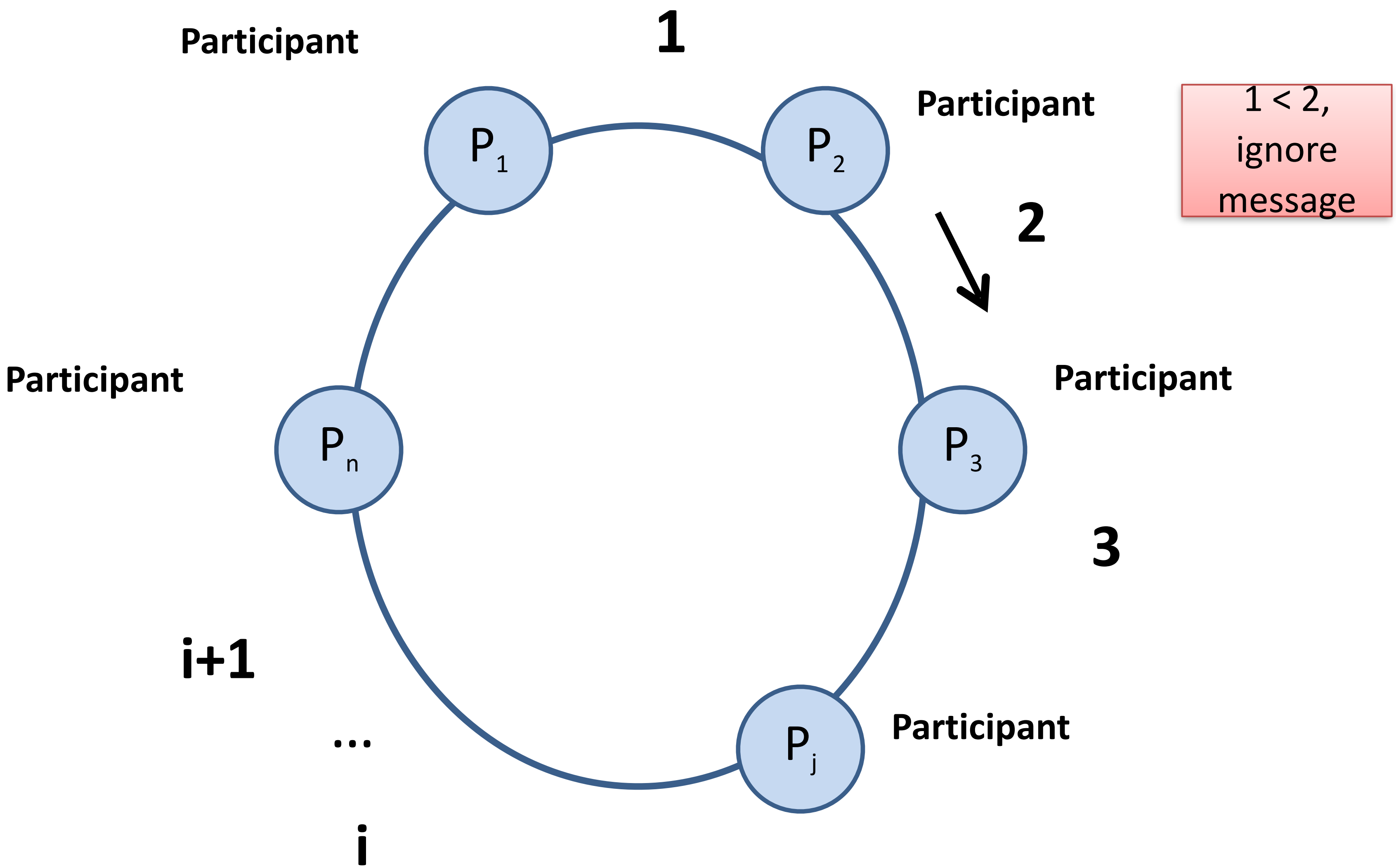
Ring-based algorithm:
Concurrent election start



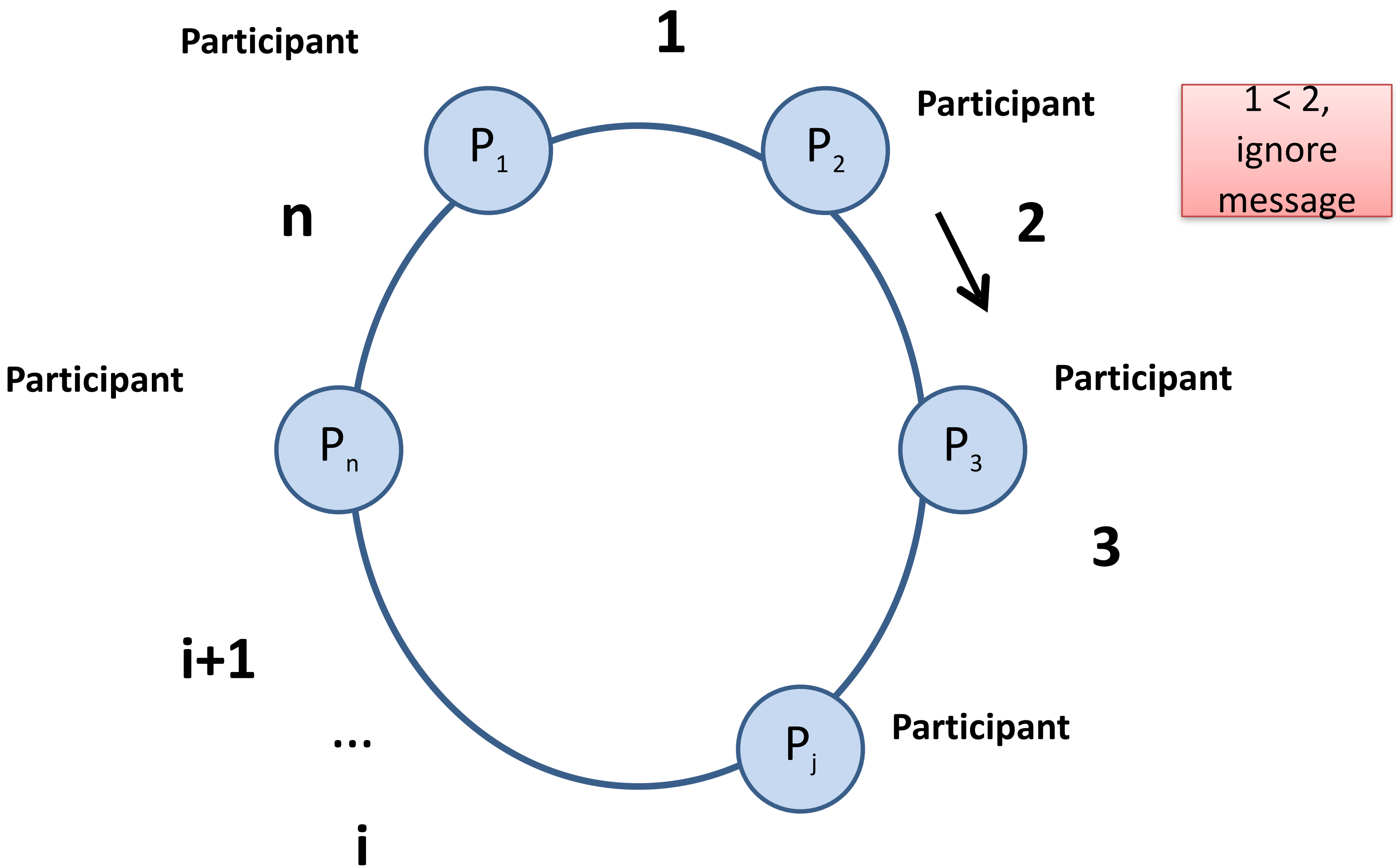
Ring-based algorithm:
Concurrent election start



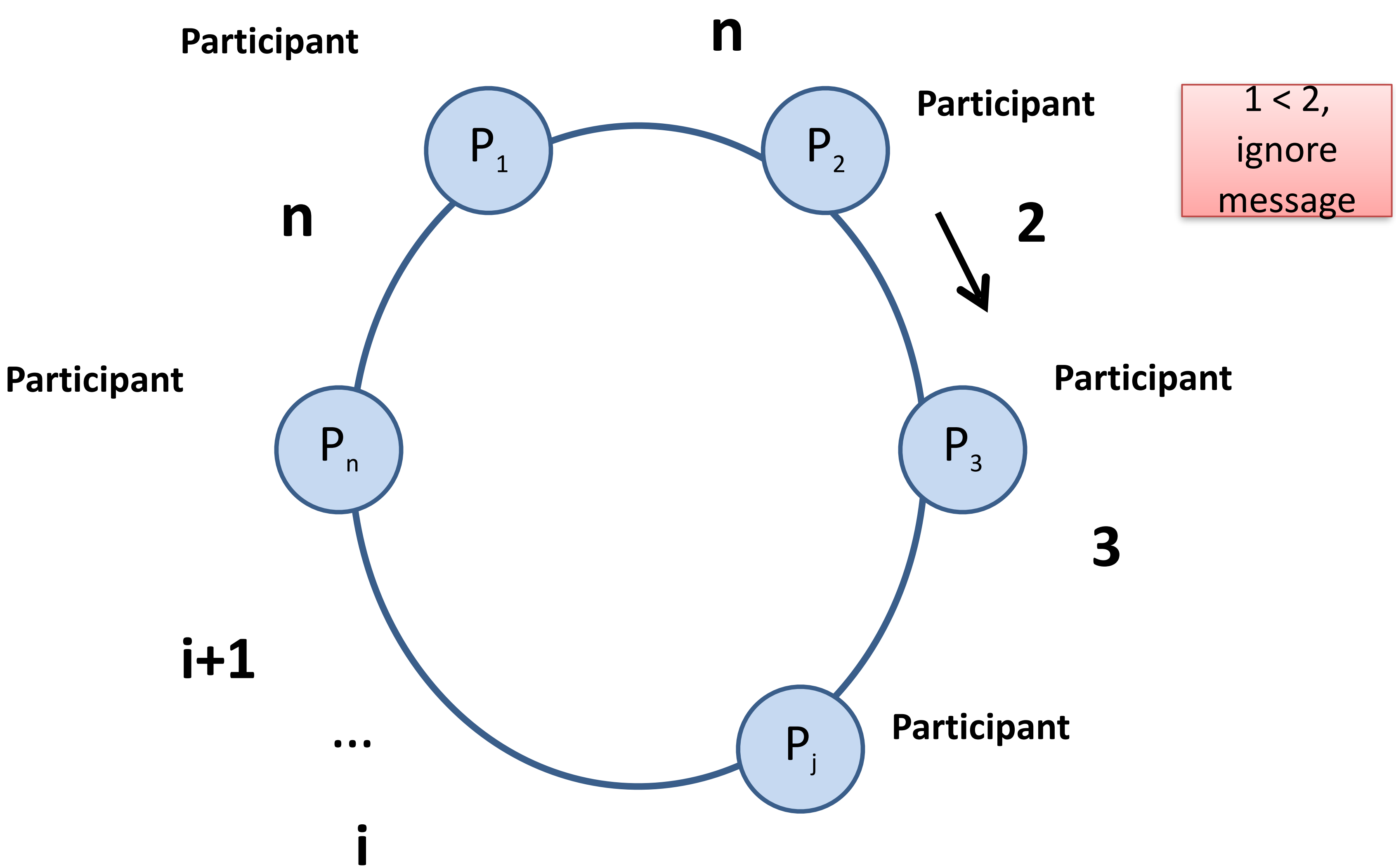
Ring-based algorithm:
Concurrent election start



Ring-based algorithm:
Concurrent election start



Ring-based algorithm:
Concurrent election start



Summarizing observations

- **Worst case**: 3N -1 messages
  - *N-1* messages to reach highest ID from lowest ID
  - *N* messages to reach point of origin
  - Leader announcement takes another *N* messages
- **Safety**: even with multiple nodes starting election
- **Liveness**: guaranteed progress *if no failures during the election occur*

Self-study questions

- Develop a mutual exclusion algorithm by reducing the problem to leader election.
- How can the algorithm be made to tolerate 1, 2, … crash faults?
- How can the algorithm be made to tolerate message loss?

48 IEEE TRANSACTIONS ON COMPUTERS, VOL. C-31, NO. 1, JANUARY 1982

Elections in a Distributed Computing System

HECTOR GARCIA-MOLINA, MEMBER, IEEE

***Abstract*—After a failure occurs in a distributed computing system, it is often necessary to reorganize the active nodes so that they can continue to perform a useful task. The first step in such a reorganization or reconfiguration is to elect a coordinator node to manage the operation. This paper discusses such elections and reorganizations. Two types of reasonable failure environments are studied. For each environment assertions which define the meaning of an election are presented. An election algorithm which satisfies the assertions is presented for each environment.**

***Index Terms*—Crash recovery, distributed computing systems, elections, failures, mutual exclusion, reorganization.**

I. Introduction

A DISTRIBUTED system is a collection of autonomous computing nodes which can communicate with each other and which cooperate on a common goal or task [4]. For example, the goal may be to provide the user with a database management system, and in this case the distributed system is called a distributed database [16].

When a node fails or when the communication subsystem which allows nodes to communicate fails, it is usually necessary for the nodes to adapt to the new conditions so that they may continue working on their joint goal. For example, consider a collection of nodes which are processing sensory data and trying to locate a moving object [18]. Each node has some sensors which provide it with a local view of the world. The nodes exchange data and together decide where the object is located. If one of the nodes ceases to operate, the remaining nodes should recognize this and modify their strategy for locating the object. A node which neighbors the failed node could try to collect sensory data for the area which was assigned to the failed node. Another alternative would be for the remaining nodes to use a detection algorithm which is not very sensitive to "holes" in the area being studied. Or the nodes could decide to switch to such an algorithm when the failure occurs. If enough nodes fail, the remaining nodes may decide that they just cannot perform the assigned task, and may select a new or better suited job for themselves.

There are at least two basic strategies by which a distributed system can adapt to failures. One strategy is to have software which can operate continuously and correctly as failures occur and are repaired [9]. (In the previous example, this would correspond to using an algorithm which can detect the object even when there are holes in the data.) The second alternative is to temporarily halt normal operation and to take some time out to *reorganize* the system. During the reorganization period, the status of the system components can be evaluated, any pending work can either be finished or discarded, and new algorithms (and possibly a new task) that are tailored to the current situation can be selected. The reorganization of the system is managed by a *single* node called the *coordinator*. (Having more than one node attempting to reorganize will lead to serious confusion.) So as a first step in any reorganization, the operating or active nodes must *elect* a coordinator. It is precisely these elections we wish to study in this paper.

In this paper we will not study the first strategy of continuous operation. This does not mean that we think that the second strategy of reorganization is superior. Which strategy is best depends on the requirements of the application and on the failure rate. If failures are very common, it will probably not pay off to reorganize. If it is not possible to stop performing the task for even a few seconds (or milliseconds?), then clearly continuous operation is a necessity. On the other hand, algorithms for continuous system operation will in all likelihood be more complex than algorithms that can halt when a failure is encountered. Thus, operation between failures should be more efficient if reorganizations are allowed.

In this paper we discuss election protocols in the context of failures, but notice that election protocols can also be used to start up a system initially, or to add or remove nodes from the system [11]. Thus, when the nodes in the system are initially turned on, they will automatically elect a coordinator and start operating, just as if they were recovering from a failure.

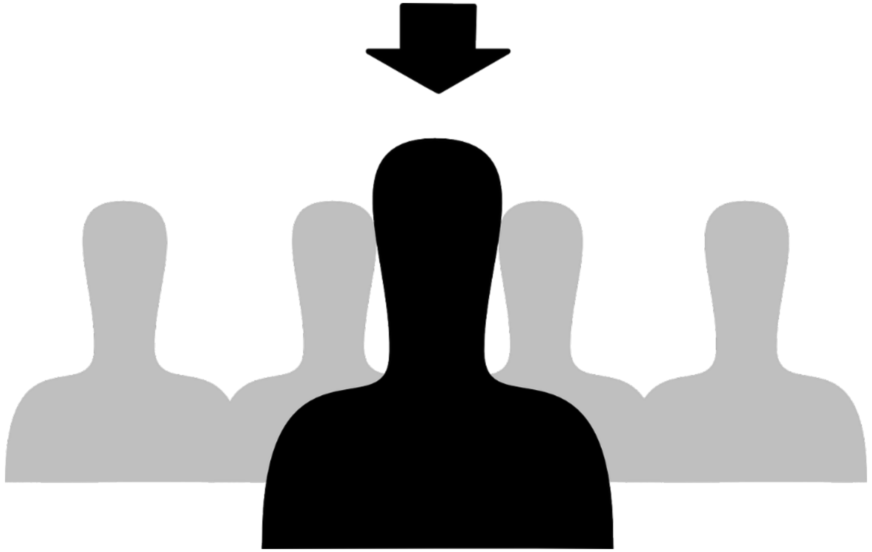
The intuitive idea of an election is very natural. We have a number of nodes which talk to each other. After some deliberation among the nodes, a single node is elected the coordinator. When the election terminates, there is only one node that calls itself the coordinator, and all other nodes know the identity of the coordinator. After the election, the coordinator can start the reorganization of the system, after which normal operation can resume.

However, when one attempts to translate the natural idea of an election to a concrete algorithm for performing the election, several interesting issues arise. For example, notice that after a node is elected coordinator, some or all of its constituents may fail. So what does it mean to be coordinator if you really cannot tell who you are coordinating? How can the election protocol cope with failures during the election itself? When certain types of failures occur, it may be impossible to guarantee that a single node calls itself a coordinator. How can these cases be handled? Furthermore, in some situations (like after the communication subsystem is partitioned) we may wish to have more than one coordinator.

Manuscript received January 7, 1981; revised July 17, 1981. This work was supported in part by the National Science Foundation under Grant ECS-8019393.

The author is with the Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ 08544.

0018-9340/82/0100-0048$00.75



Pixabay license

LEADER ELECTION

BULLY ALGORITHM, 1982

Bully algorithm, 1982

- Assumes each node has a **unique ID** (UID), reliable message delivery and synchronous system
- Assumes nodes know each others' UIDs and can directly communicate with one another
  - Higher UIDs have priority
  - Can "bully" nodes with lower UIDs
- Initiated by any node that **detects** leader failure
- Tolerates nodes crashing, even during elections
- Crash recovery model

Bully algorithm messages

- ***Election*** message
  - announces an election
- ***Answer*** message
  - responds to an election message
- ***Coordination*** message
  - announces victory and identifies as leader

@ P_i

P_i detects failure of leader

For any $j < i$ and any $i < k$

@ P_i — P_i detects failure of leader

For any $j < i$ and any $i < k$

1. Broadcasts **election message** to all nodes k with $k>i$

@ P_i

P_i detects failure of leader

For any j **<** i and any i **<** k

1. Broadcasts **election message** to all nodes k with $k>i$
2. Any P_k receiving election message **responds with answer message** and starts another election

@ P_i

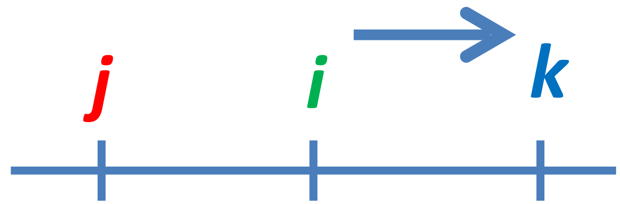
P_i detects failure of leader

For any $j < i$ and any $i < k$

1. Broadcasts **election message** to all nodes k with $k>i$
2. Any P_k receiving election message **responds with answer message** and starts another election
3. Any P_j receiving election message **does not respond**

@ P_i

P_i detects failure of leader



For any $j < i$ and any $i < k$

1. Broadcasts **election message** to all nodes *k* with $k>i$
2. Any P_k receiving election message **responds with answer message** and starts another election
3. Any P_j receiving election message **does not respond**
4. If P_i **does not receive any answer message (timeout)** then it **broadcasts the coordination message**

@ P_i

P_i detects failure of leader

For any $j < i$ and any $i < k$

1. Broadcasts **election message** to all nodes *k* with *k>i*
2. Any P_k receiving election message **responds with answer message** and starts another election
3. Any P_j receiving election message **does not respond**
4. If P_i **does not receive any answer message (timeout)** then it **broadcasts the coordination message**
5. If P_i **does receive answer message(s)** then waits to receive **coordination message**

@ P_i

P_i detects failure of leader

For any $j < i$ and any $i < k$

1. Broadcasts **election message** to all nodes k with $k>i$
2. Any P_k receiving election message **responds with answer message** and starts another election
3. Any P_j receiving election message **does not respond**
4. If P_i **does not receive any answer message (timeout)** then it **broadcasts the coordination message**
5. If P_i **does receive answer message(s)** then waits to receive **coordination message**
6. If P_i receives no coordination message before timeout, **it restarts an election**

@ P_i P_i detects failure of leader

Special case $i = k$

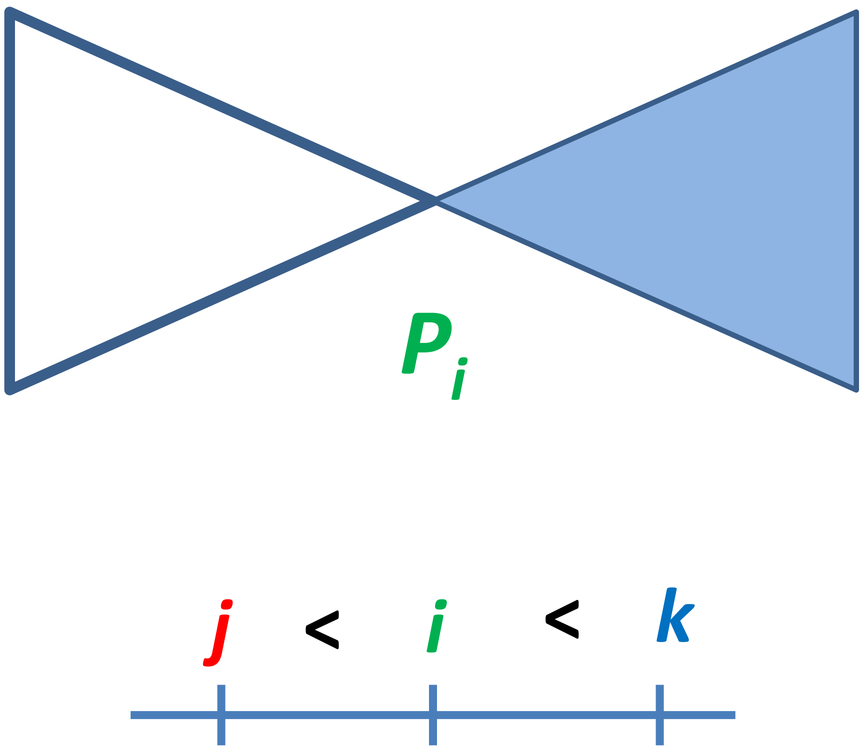
For any $j < i$ and any $i < k$

…

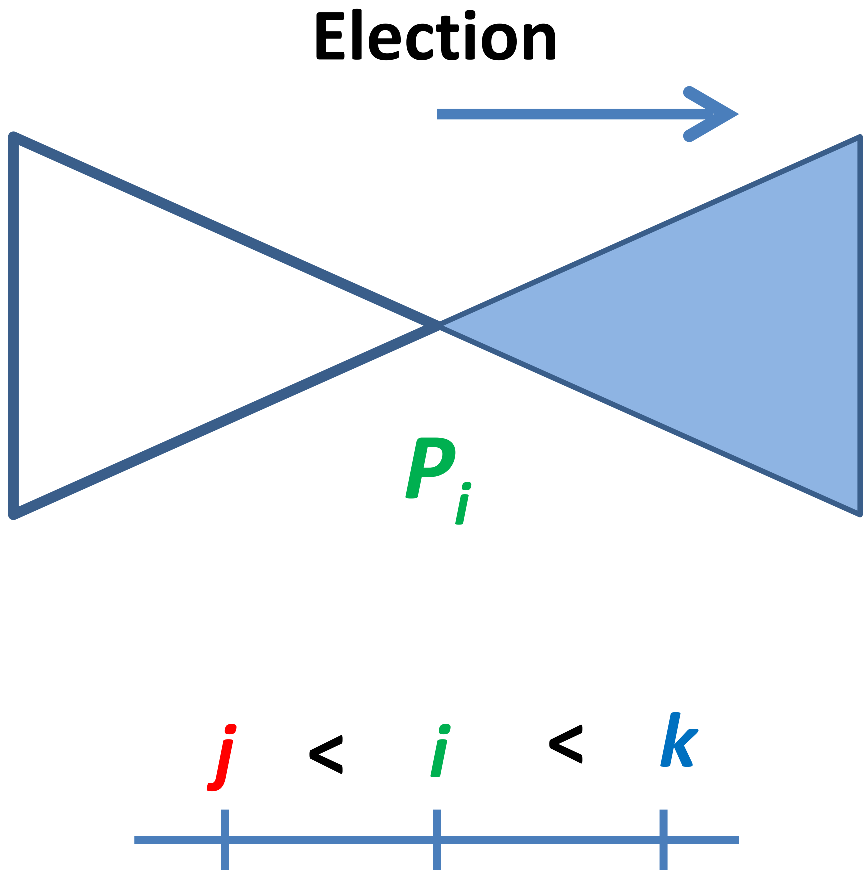
4. If P_i **does not receive any answer message (timeout)** then it **broadcasts victory** via **coordination message**

Therefore, if P_i knows it has the highest UID (e.g., after crash recovery), it may as well directly declare itself a leader (sending out the coordinator message).

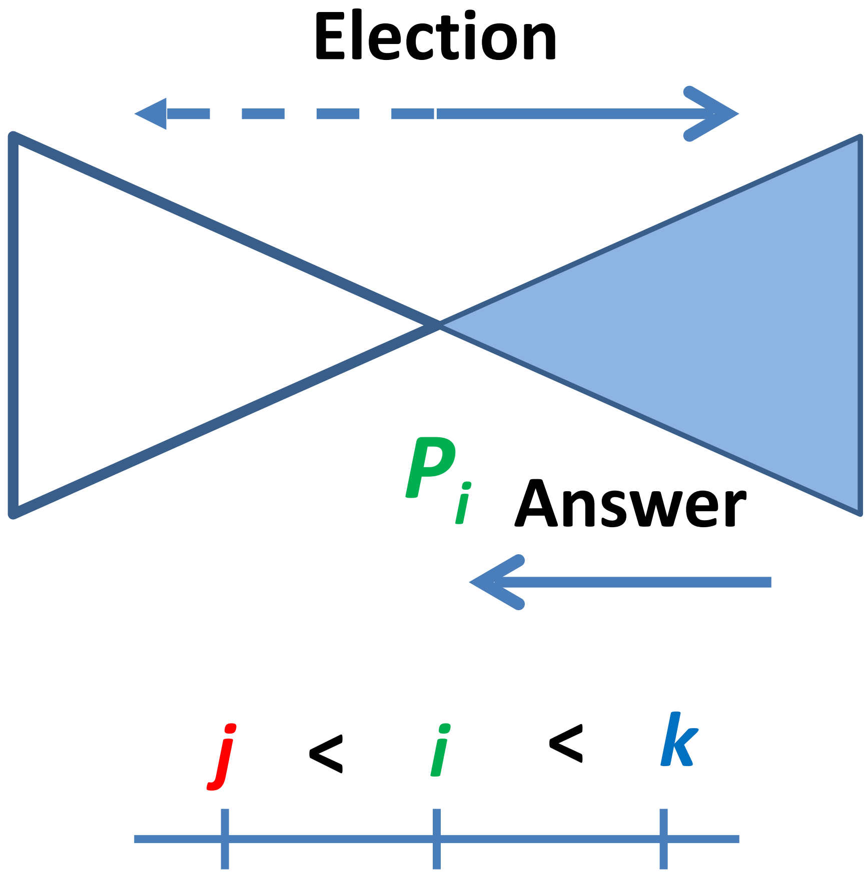
Election and answer message flow



Election and answer message flow



Election and answer message flow



Why Bully?

Upon node (leader) crash

- Suppose node eventually recovers (*no problem if it stays down, why?*)
- Node may determine that it has the highest UID, thus, it would pronounce itself as leader
  - Even though system may have an existing leader (elected during crash episode)
- New node "**bullies**" current leader

Summary: Safety and liveness

- **Safety** – argue by contradiction
  - Assume two leaders $\mathbf{P_i}$, $\mathbf{P_k}$ ($i \neq k$)
  - Thus, $\mathbf{P_i}$, $\mathbf{P_k}$ exchanged victory messages
  - Thus, $\mathbf{P_i}$, $\mathbf{P_k}$ exchanged election messages
  - But node with lower UID would not have replied
- **Liveness**
  - Would-be leader (*WbL*) fails after sending answer but before sending coordination message
  - Depending on timeout period of nodes with lower UIDs
    - If *WbL* recovers, it issues coordination message
    - If *WbL* does not recover, a new election materializes a leader

Self-study question

- What are the pros and cons of broadcasting election messages to all nodes vs. to only broadcasting them to nodes with higher UIDs?
- Is the algorithms safe during an election, explain why or why not?
- What is Bully's worst and best case message complexity, assuming *n* nodes in total?