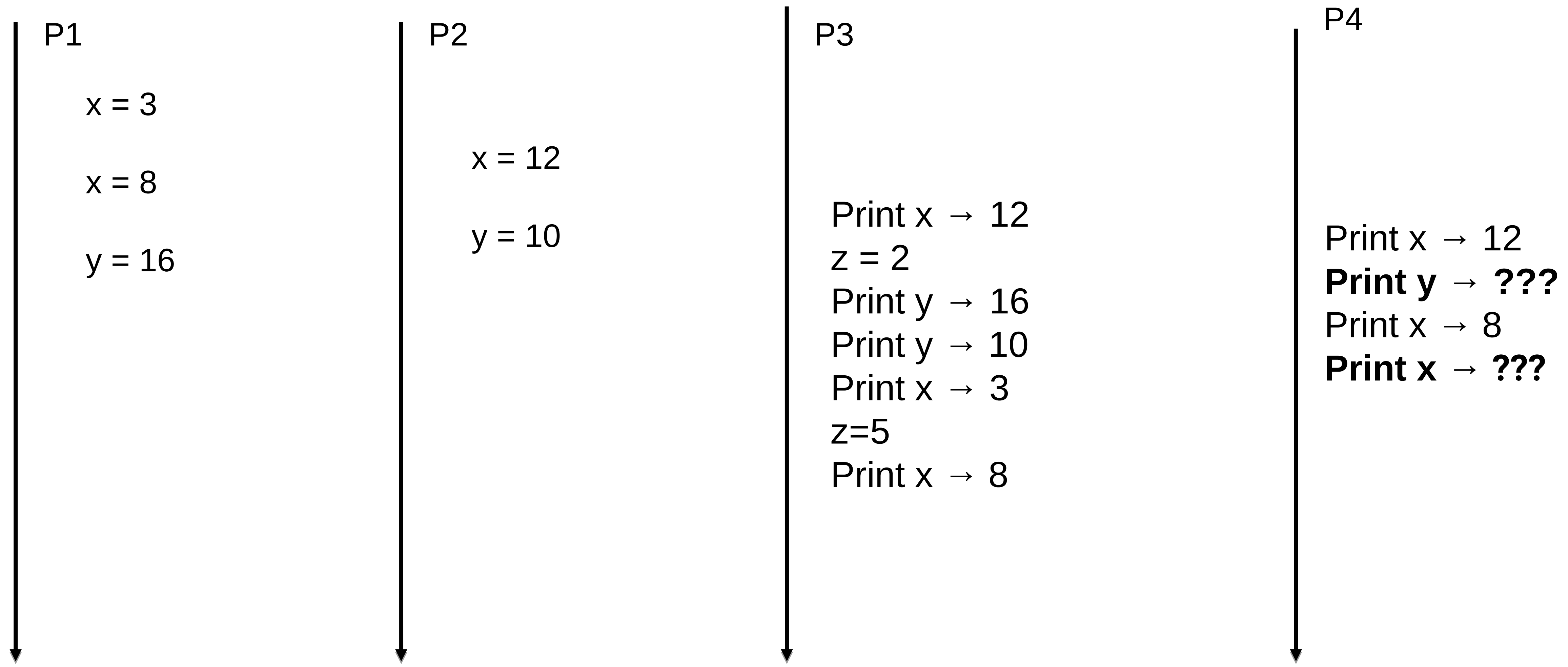
Coordination

Part-1

RECAP: **Exercise:** Sequential consistency

Fill in the question marks:

What should the read operations return to be sequentially consistent?



Coordination

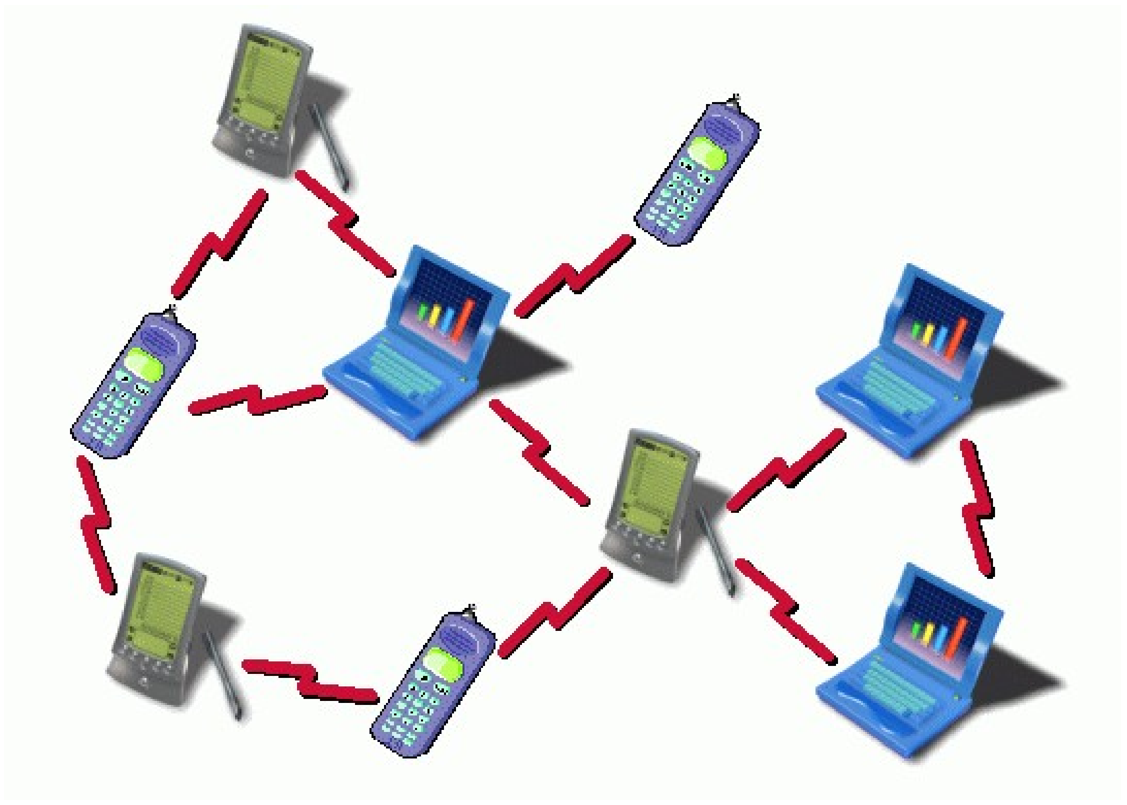
Part-1

2,000 drones light up
night sky in Shanghai to
welcome new year

General Problem

Given a set of processes Π={pi}, distributed over multiple hosts

- how to coordinate actions ?
- agree on contents of shared variables (“global state”) ?



Example problems

- control access to common database (locking)
- elect central node in ad hoc network
- elect time server in network
- avoid static master-slave relations to enhance robustness

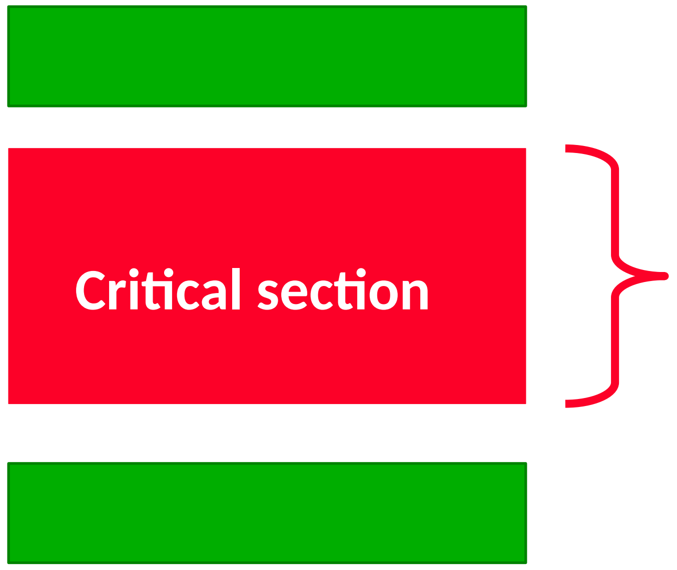
Distributed Mutual Exclusion

Election Mechanisms

1. **Distributed mutual exclusion**
    1. Problem statement
    2. Evaluation metrics
    3. Centralized approach
    4. Ring approach
    5. Multicast approach
        1. *Ricart-Agrawala*
        2. *Maekawa voting*
2. **Election**

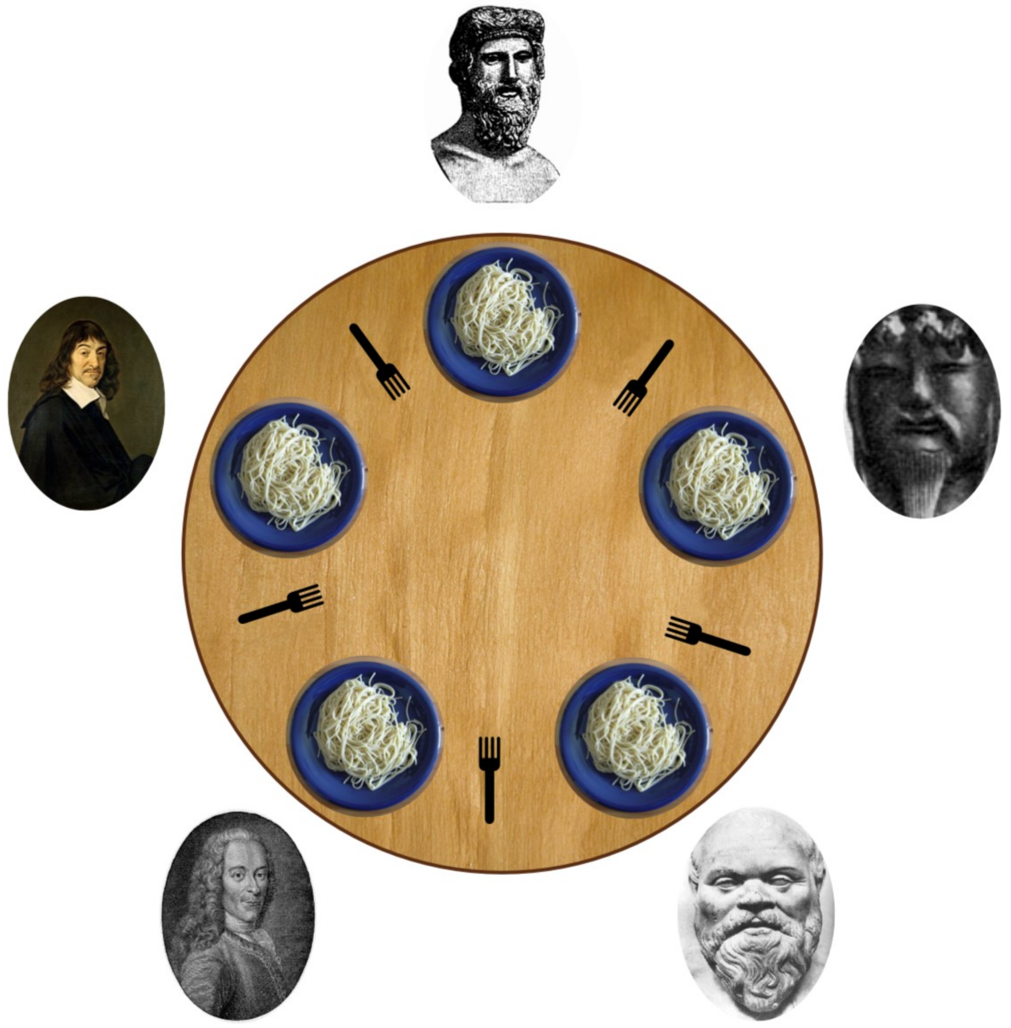
Critical sections

Goal: Coordinate process access to shared resourced



Accesses common resource

No other process should access same resource



Distributed mutual exclusion

- no shared variables between processes
- no support from common coordinating OS kernel
- only rely on message passing

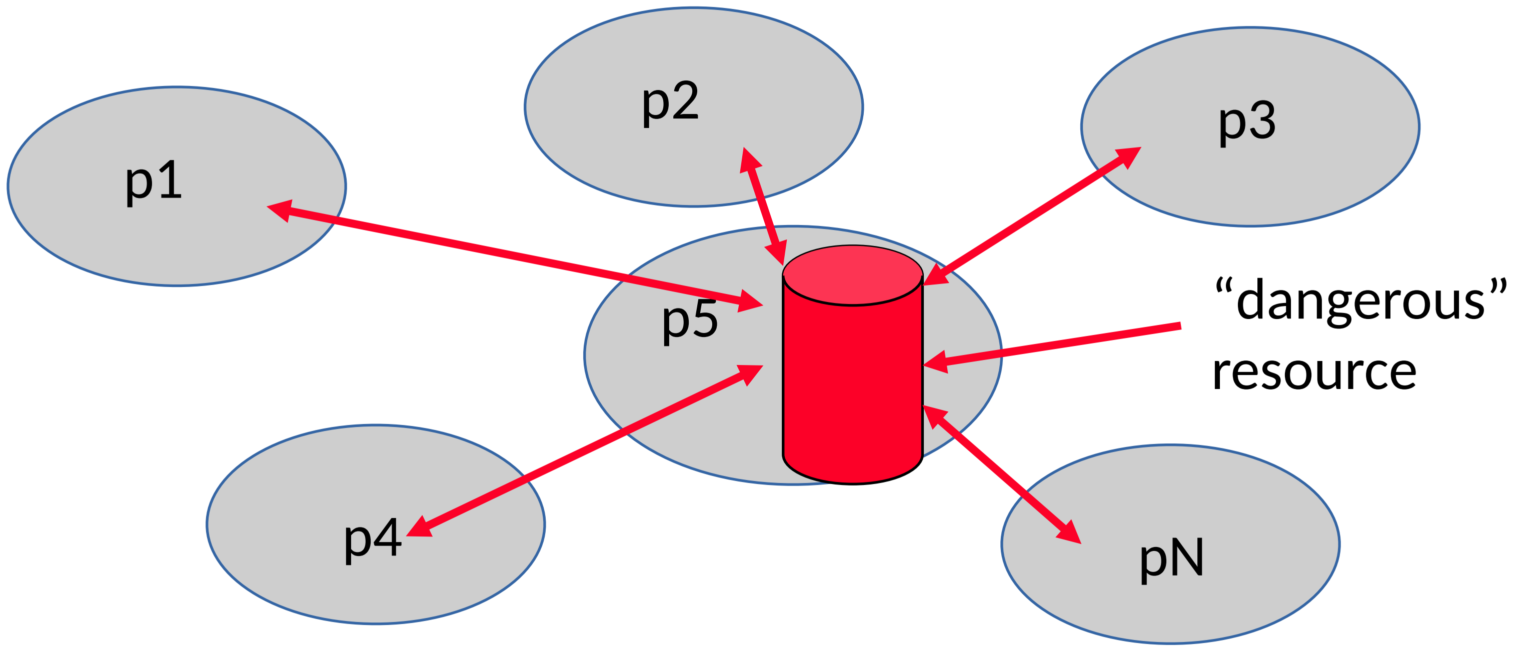
Problem statement

Consider

- **N** processes **{p1, … , pN}**, NO shared variables
- access common resources in a critical section
- asynchronous system
- processes CAN communicate (know each other)

Failure modes

- reliable channel (each message delivered, exactly once)
- no process failures
- processes are well-behaved
  (leave critical section eventually)



A good solution should …

1

Be safe [REQUIRED]

At most ONE process may execute in critical section at any time

2

Ensure liveness [REQUIRED]

- Requests to enter/leave critical sections eventually succeed
- deadlock-free algorithm
- no starvation

3

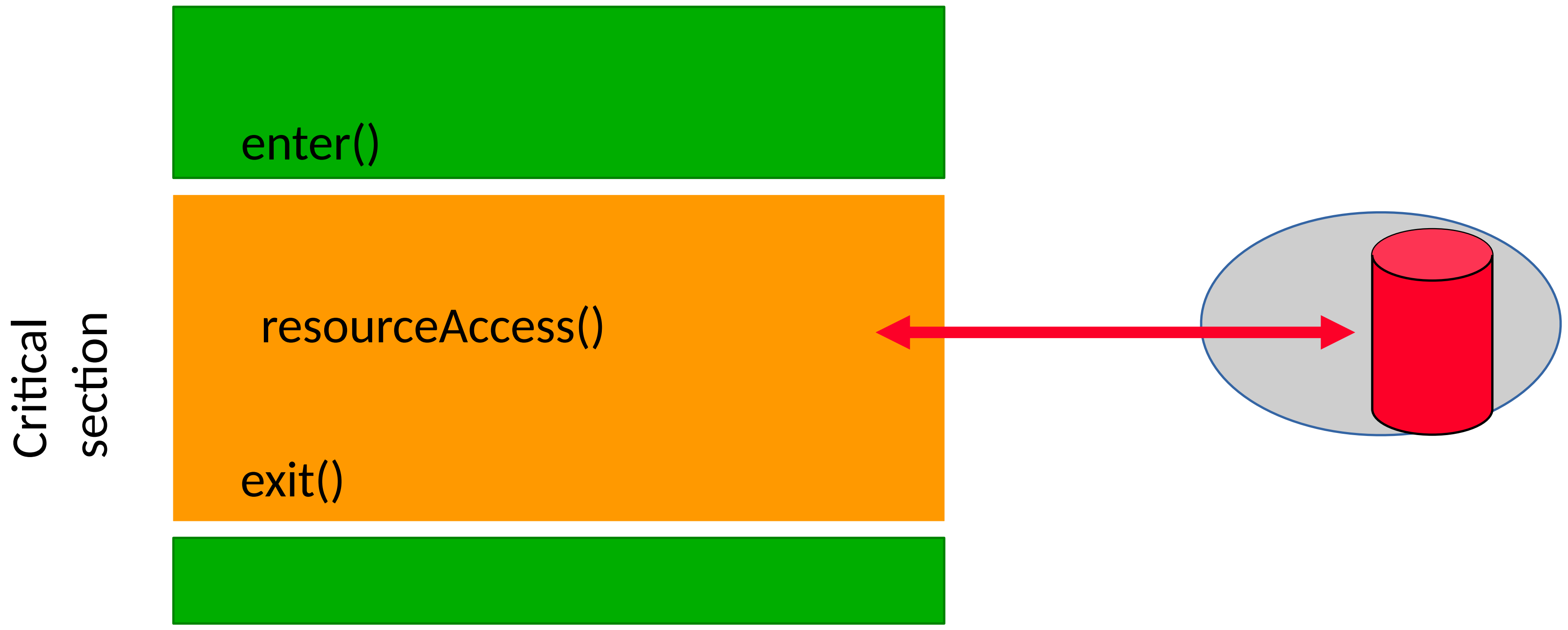
Be fair[BONUS]

Access to critical section is granted using “happened- before” relation
Thus, use logical clock to order access requests

Critical section access API

Application level primitives

enter()	enter critical section, block if necessary
resourceAccess()	access the shared resource (in critical section)
exit()	leave critical section – make free for other processes



How to quantify solution quality ?

Evaluation metrics

1

Bandwidth consumption

= number of messages sent to enter/leave critical section

2

Client delay

- time needed to enter/leave critical section
- measured in UNLOADED system
- One-way network delay

How to quantify solution quality ?

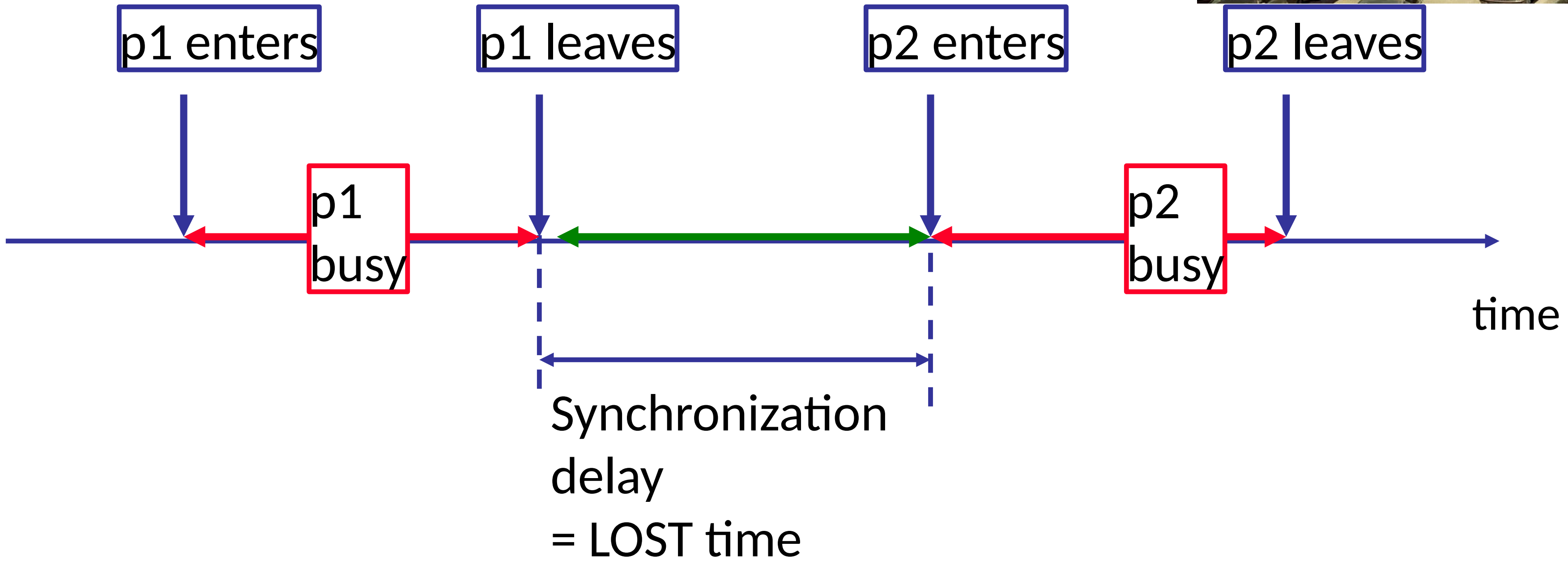
3

System throughput

- how many processes can access critical section in given time period ?
- depends on resourceAccess() time
- derived measure:

synchronization delay = average (time process (i+1) enters – time process (i) leaves)

- LOADED system

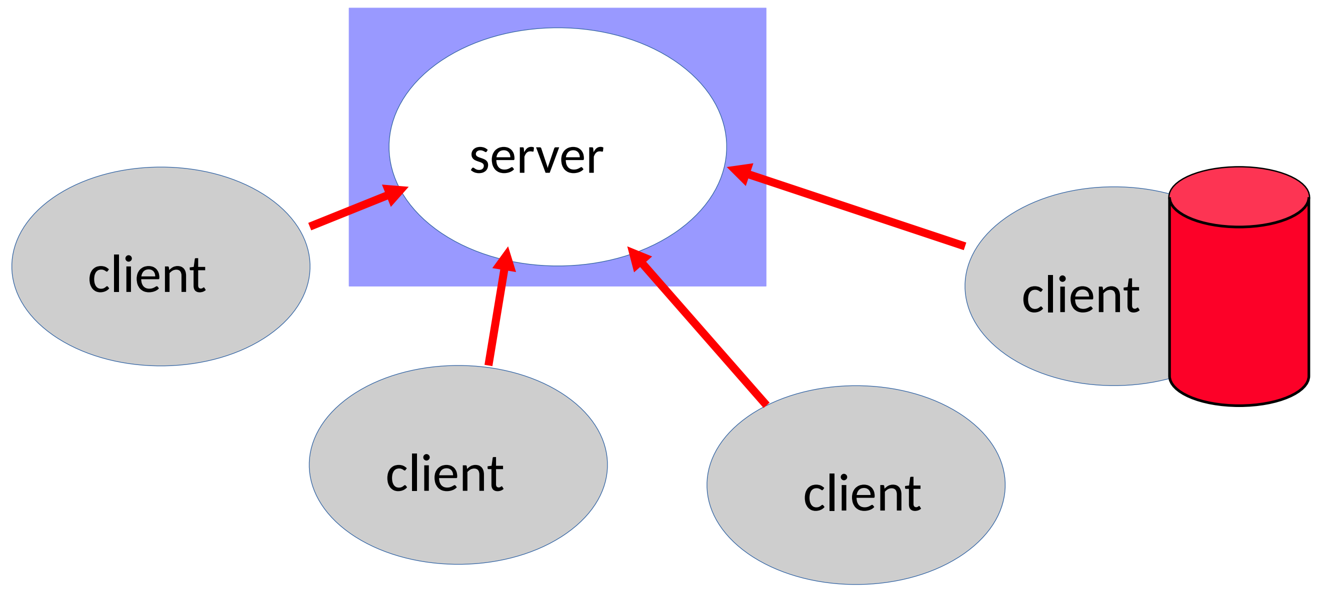


How would you build such a system?

Central server

Centralized algorithms (one server)

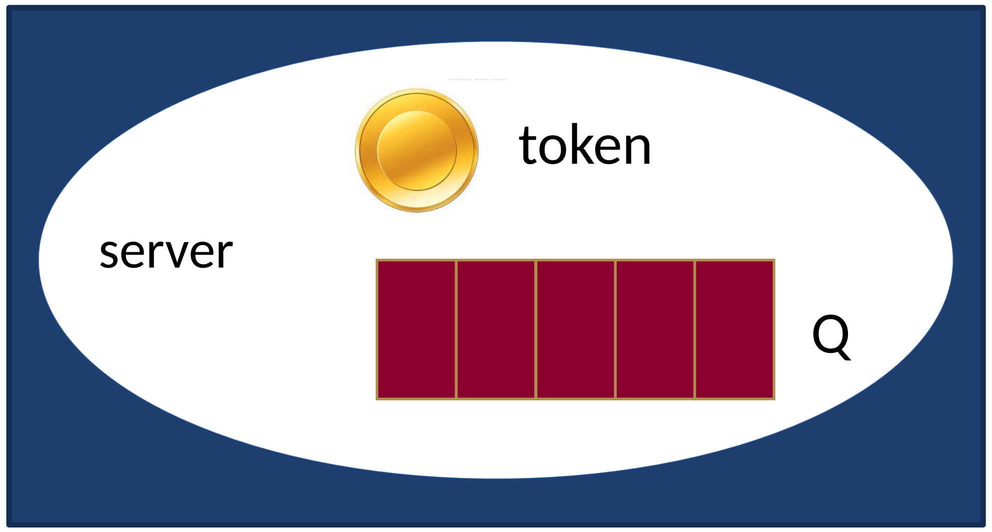
- easy
- but typically poor scaling



Messages

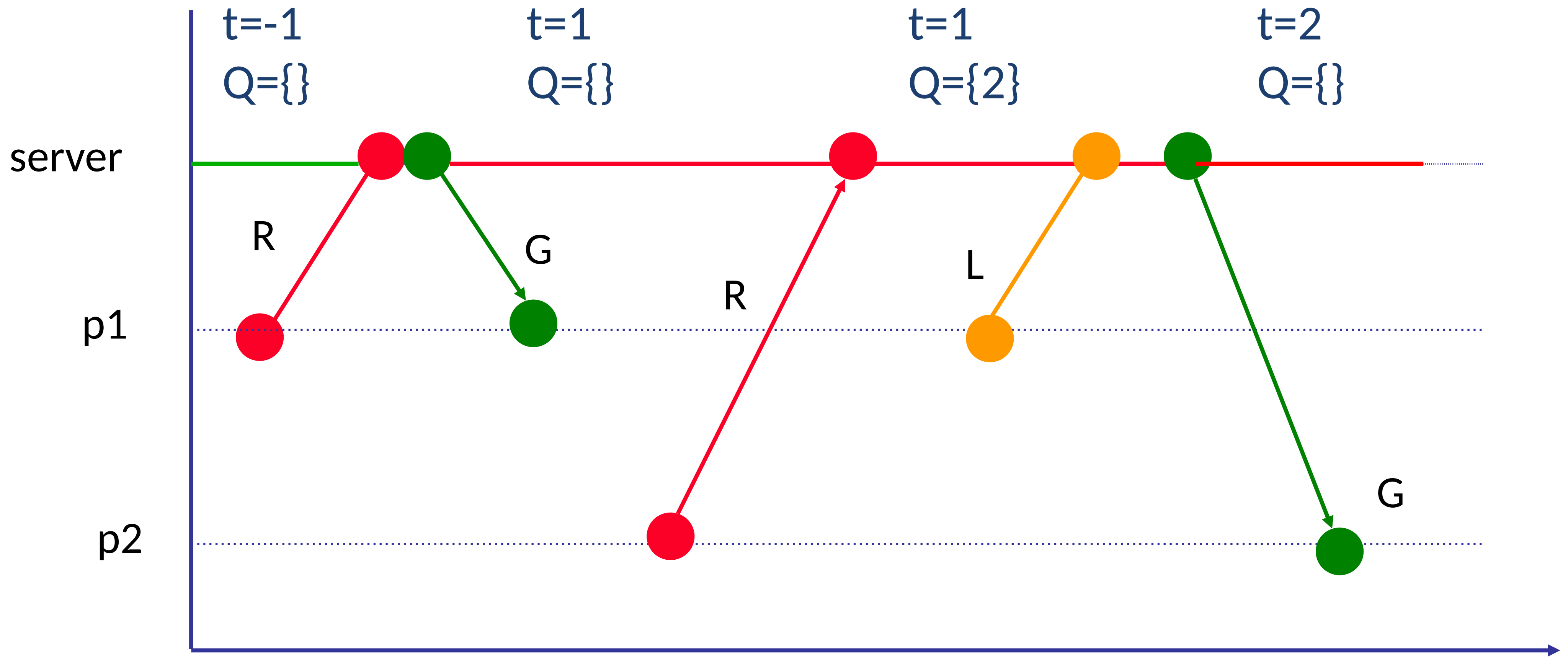
- Client -> Server : **Request**
- Server -> Client : **Grant**
- Client -> Server : **Leave**

Central server algorithm



token variable
== process ID currently active
== -1 if critical section not taken

message queue Q
stores pending requests



Central algorithm

Client side

```
1. enter()
      send Request message to server
      wait until Grant received
2. accessResource()
      perform any application specific logic
3. leave()
      send Leave message to server
```

Server side

```
When receiving Request
        if (token == -1) {
              send Grant to requesting process
              token=sender(Request)
        } else
              enqueue Request in Q
When receiving Leave
        1. dequeue oldest message m from Q
        2. token=sender(m)
        2. send Grant to sender(m)
```

Algorithm OK ?

1

Safety

guarded by token variable

✓

2

Liveness

1. Request to enter
   - all processes eventually leave
   - each leave dequeues a message from Q
   - if oldest Request dequeued
   - => every Request eventually handled

✓

2. Request to leave
   - no permission needed from server

✓

3

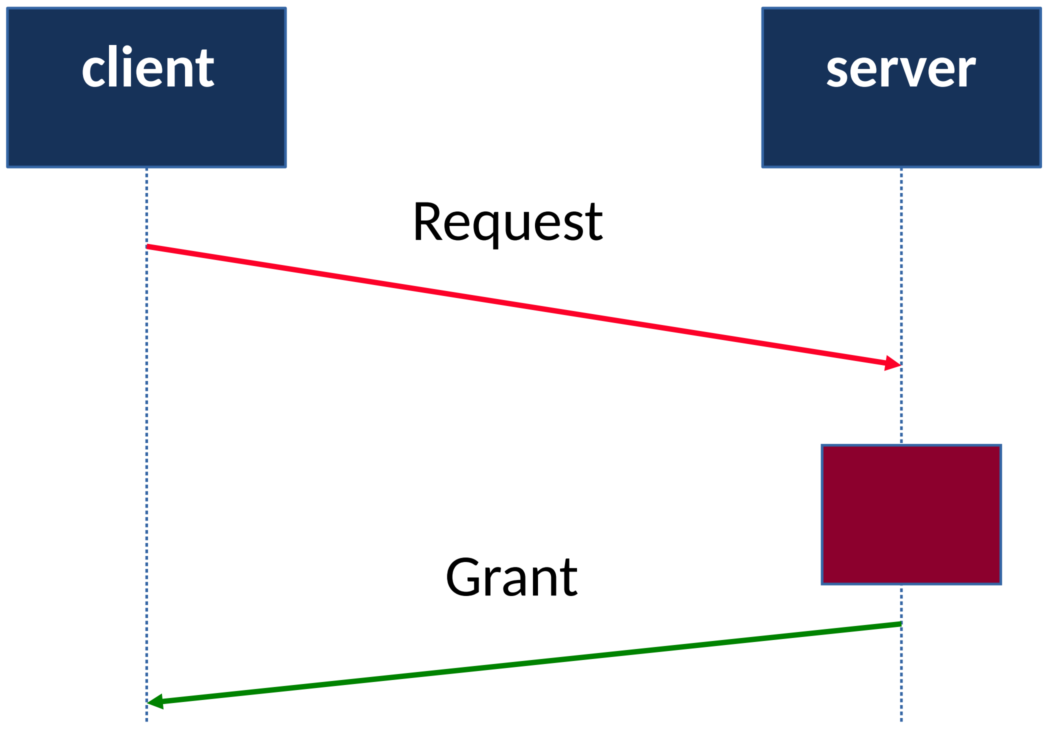
Fairness

Order Q according to “happened-before”

Algorithm efficient ?

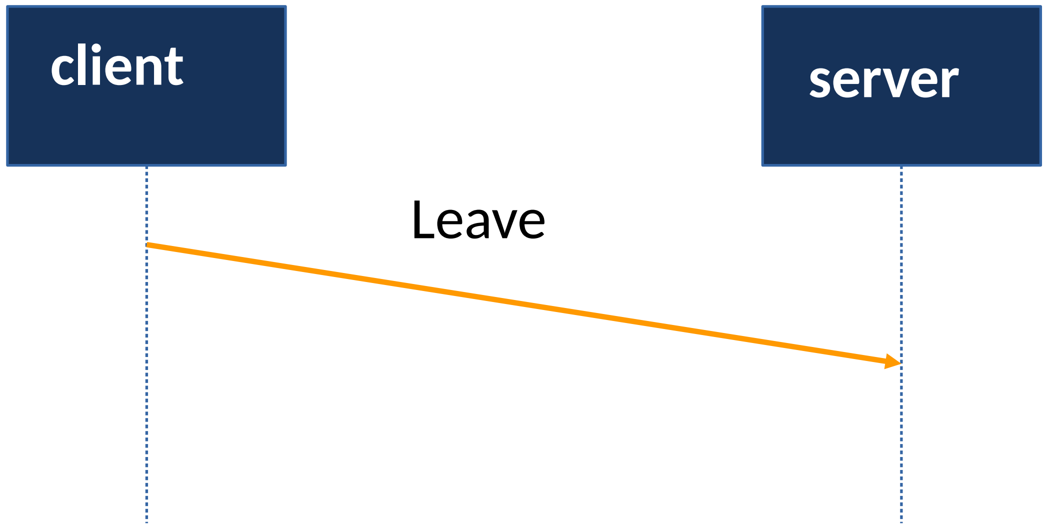
Bandwidth usage

enter()



=> 2 messages/enter

leave()



=> 1 message/leave

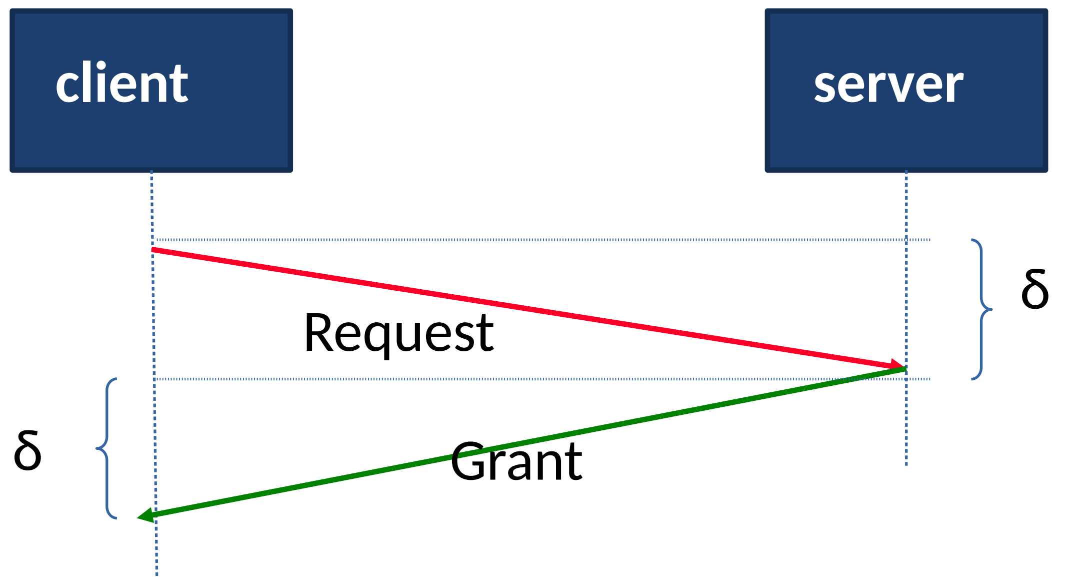
Algorithm efficient ?

Client delay (unloaded system)

δ=time needed for 1 communication

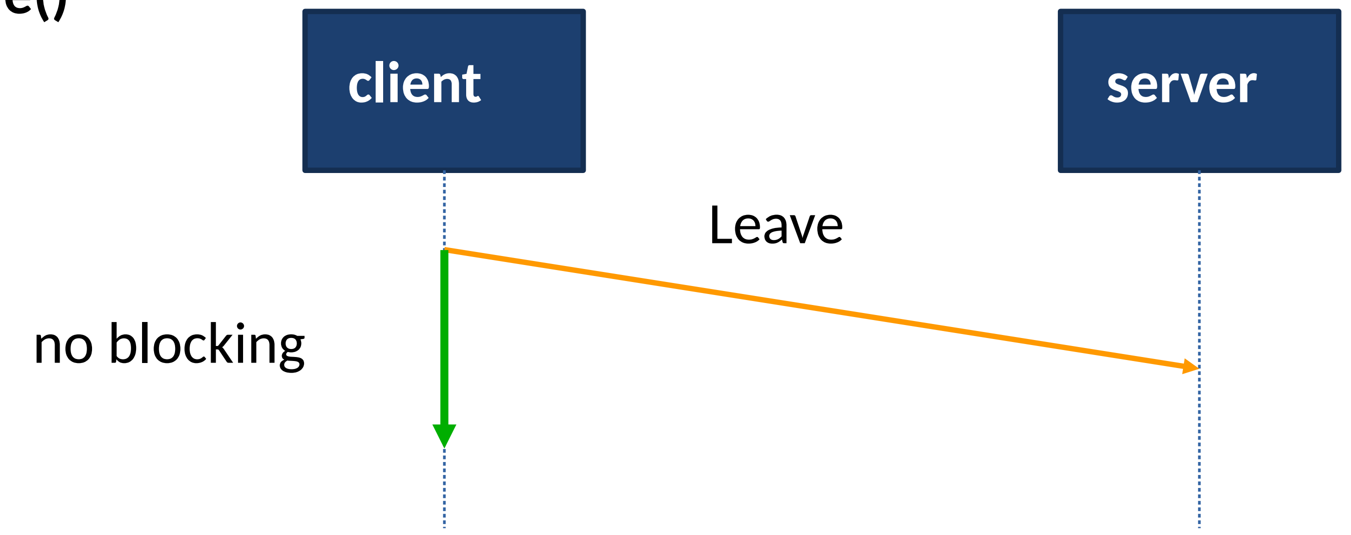
RTT = 2δ

enter()



=> 2δ

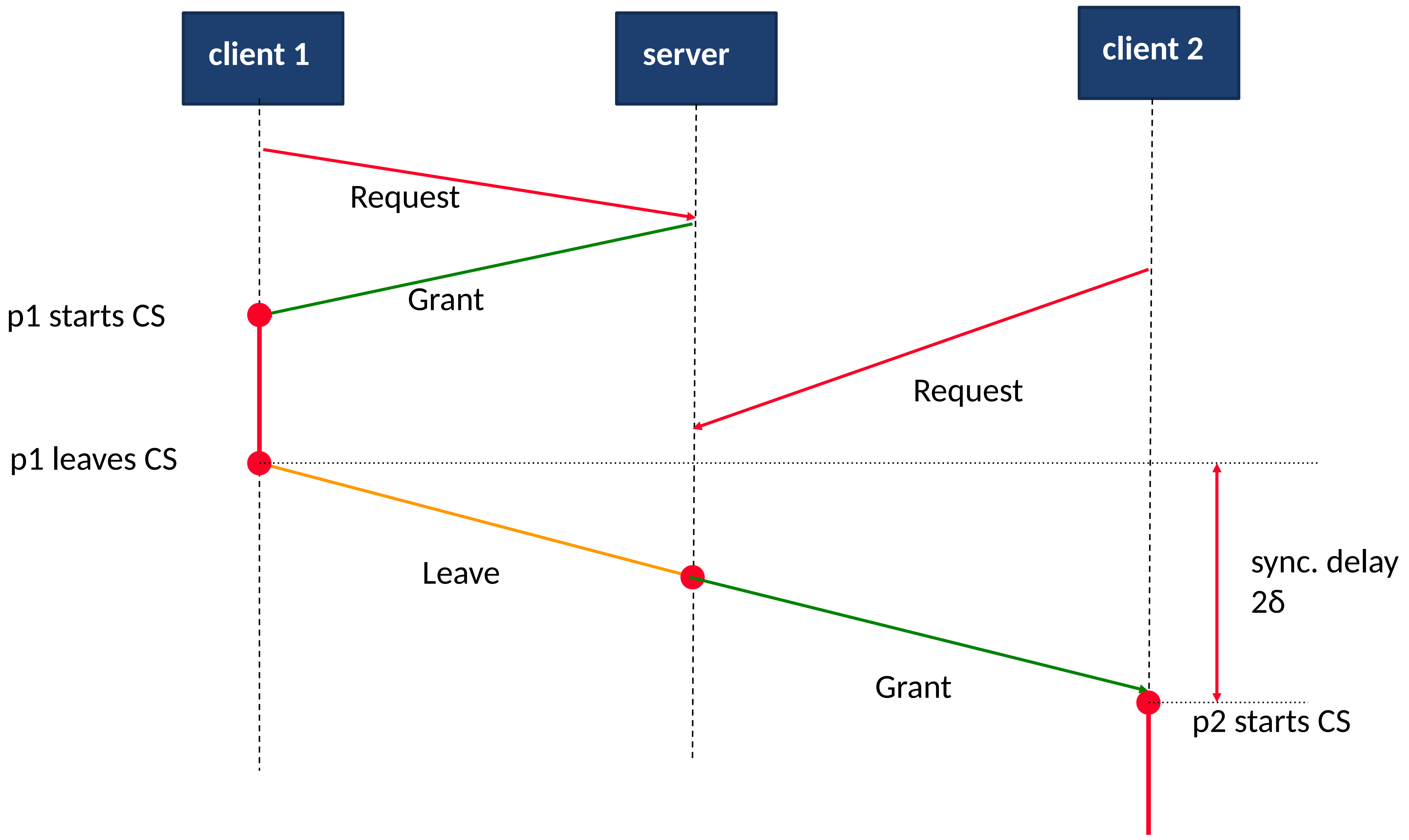
leave()



=> 0δ

Algorithm efficient ?

Synchronization delay (loaded system)

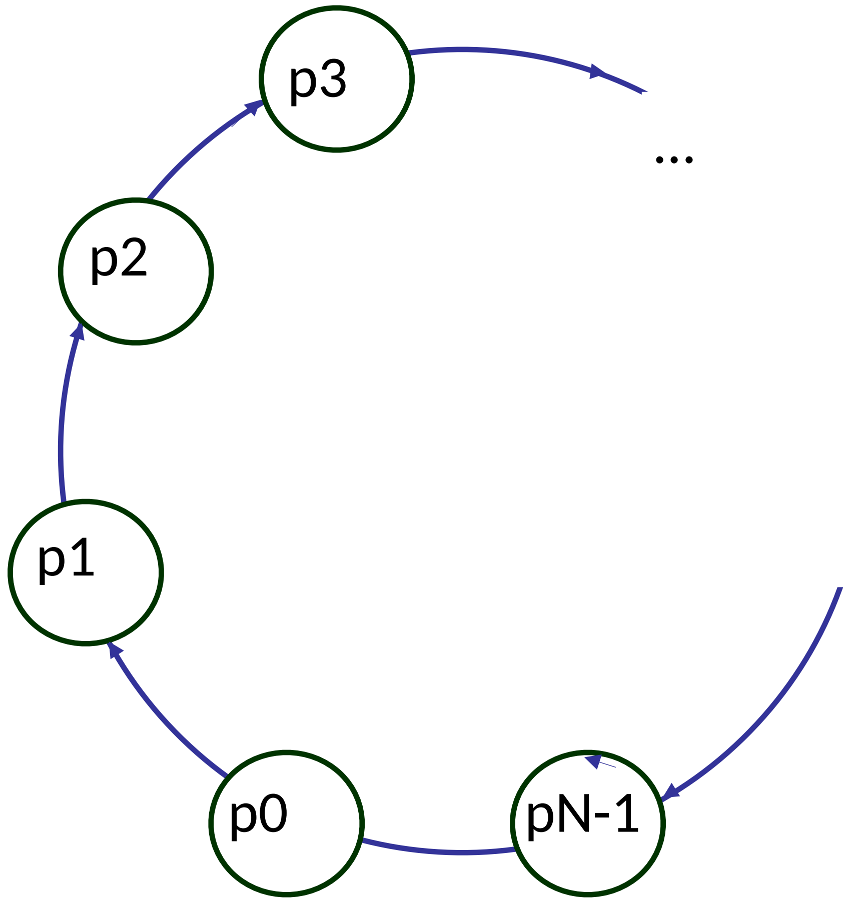


Ring-based algorithm

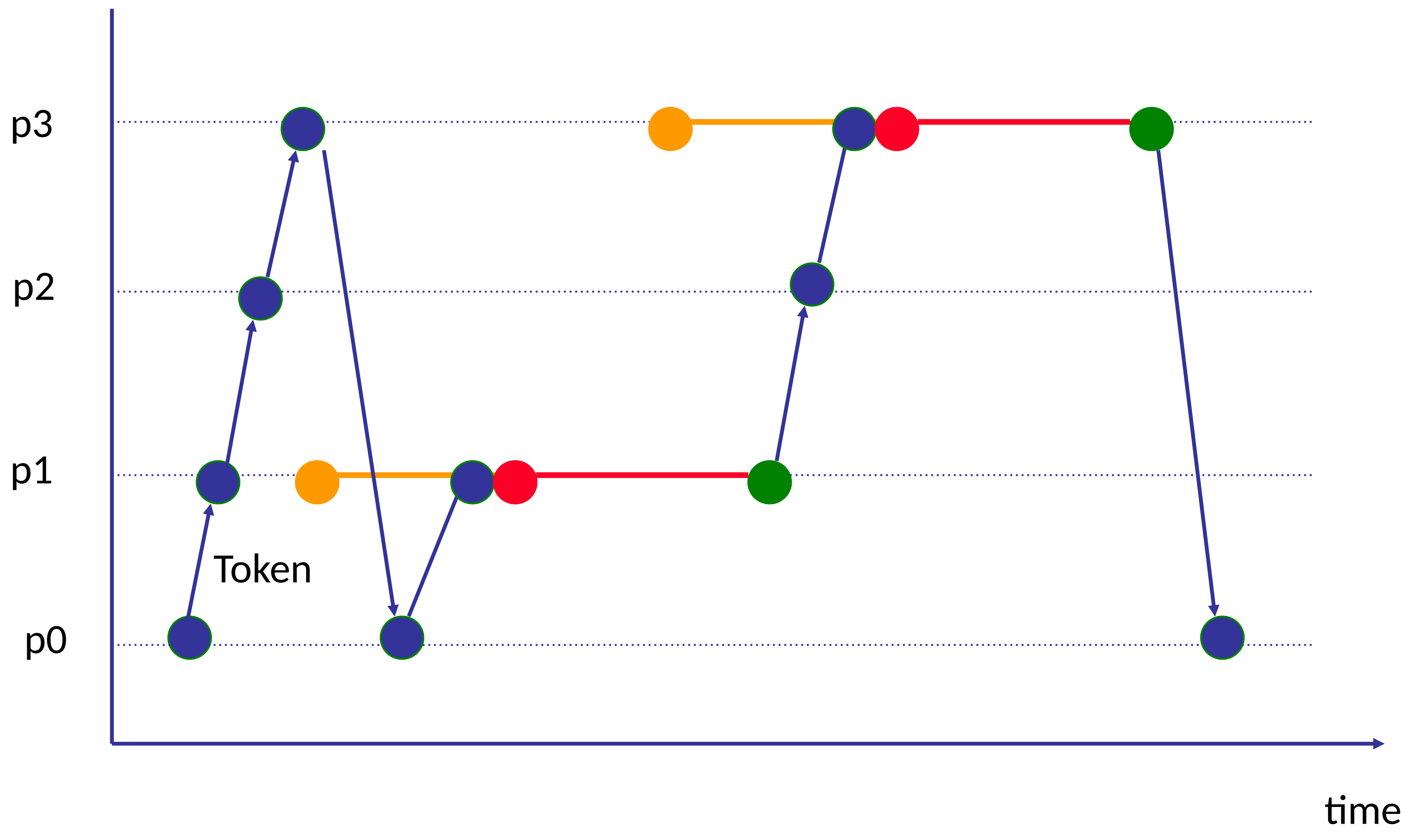
- Processes {$\mathbf{p_0}$, **…**, $\mathbf{p_{N-1}}$} arranged in logical ring
- Process $\mathbf{p_i}$ has one unidirectional communication channel to $\mathbf{p_{(i+1)}}$
- token = message passed along ring

Only one process has token
Symmetric algorithm
(no "special" process)

Message : Token



Ring-based algorithm



● process wants critical section

● process executes critical section

● process leaves critical section

Algorithm OK ?

each process p

```
When process p receives "Token"
        if (p wants access) {
            execute logic in critical section
            leave()
        } else send(Token) to next process
```

1

Safety

process can only send Token if it has received Token ✓

2

Liveness

process eventually leave

=> Token circulates in the ring

(p not allowed new access !)

=> No starvation ✓

3

Fairness

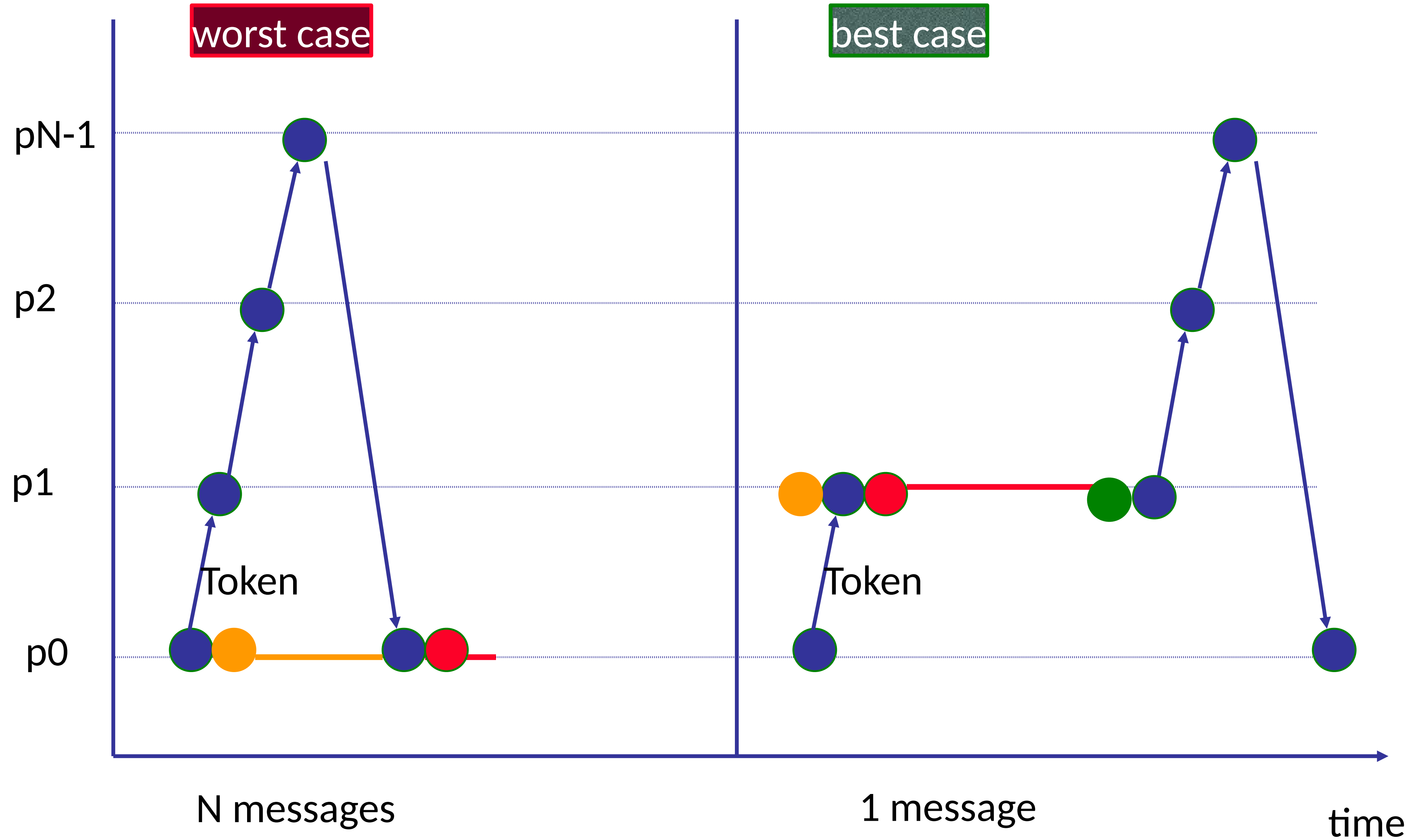
not guaranteed : processes need luck for early access

access order not based "happened before" of requests 🚫

Algorithm efficient ?

Bandwidth

enter



leave : included in enter

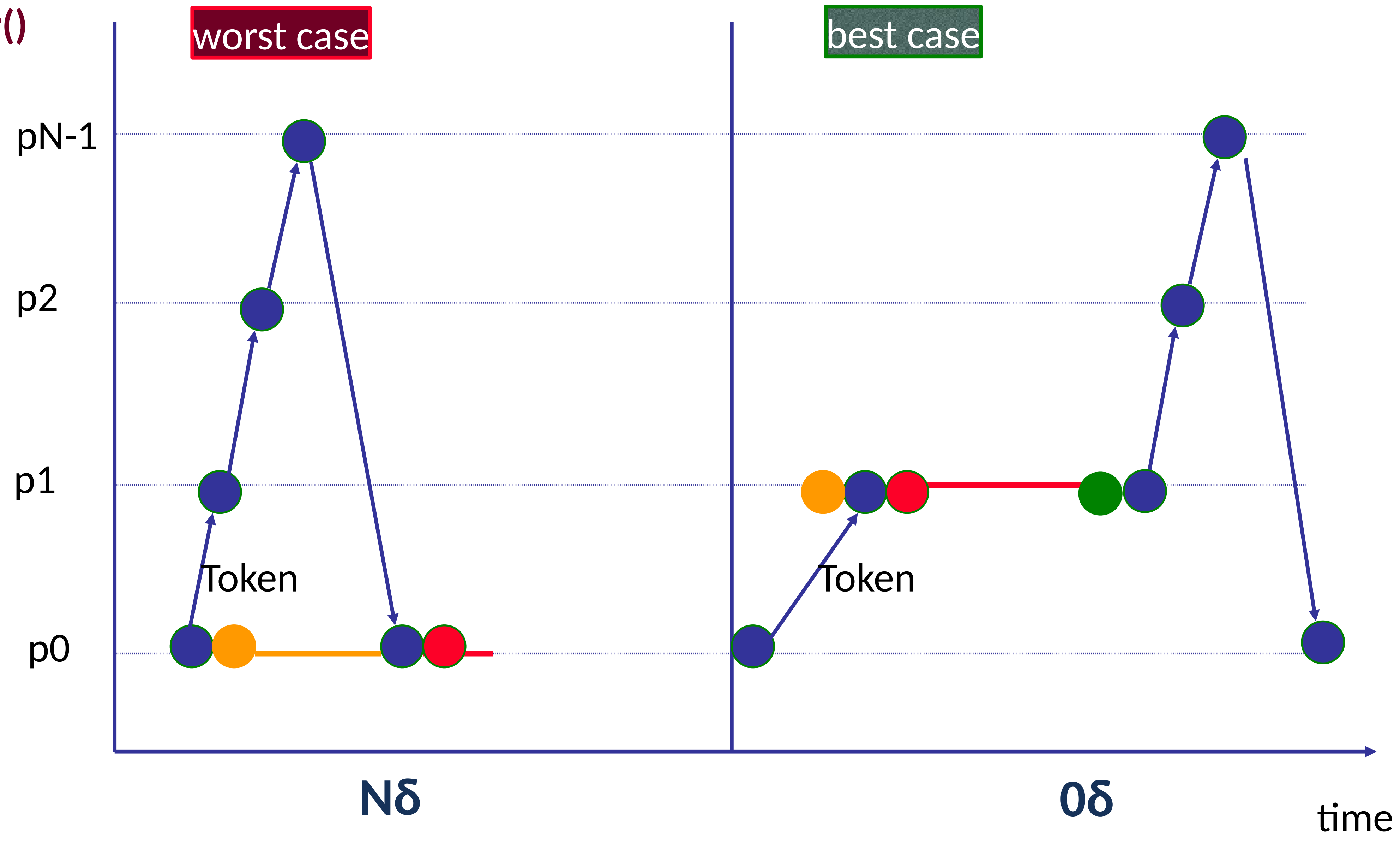
BUT : algorithm always consumes bandwidth !

average : (N+1)/2 messages

Algorithm efficient ?

Client delay (unloaded system)

enter()

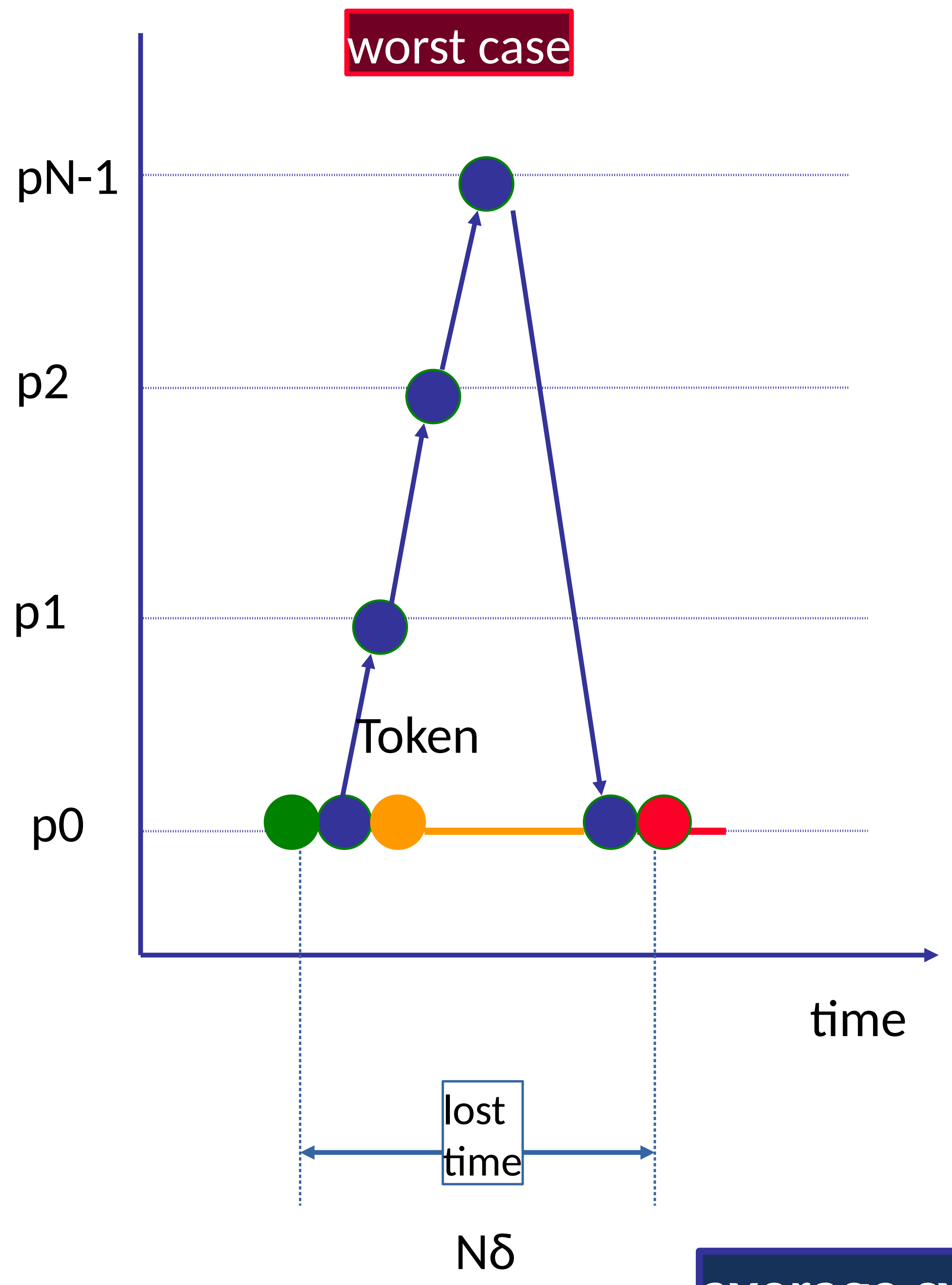


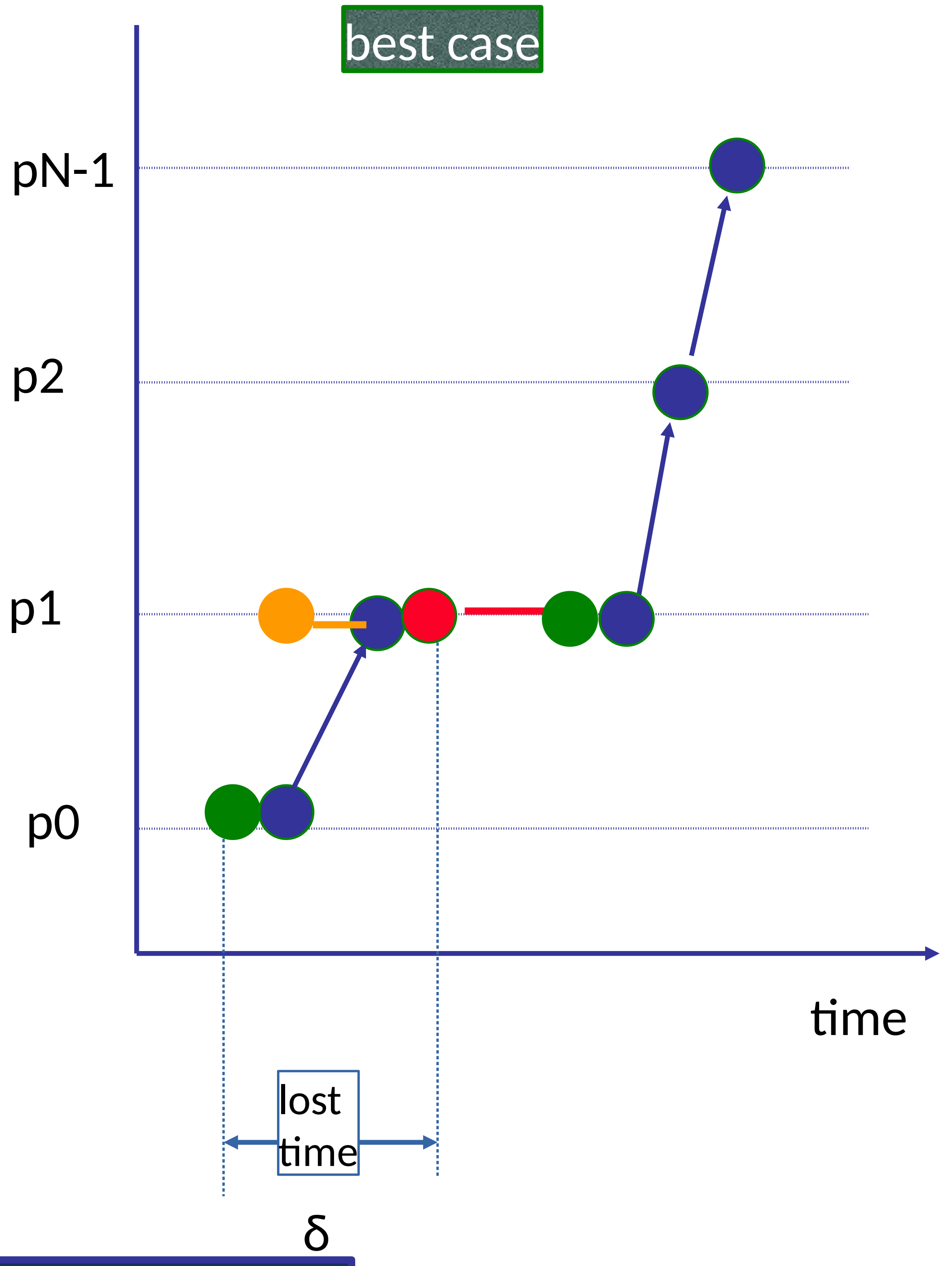
leave() : no blocking for process to leave critical section 0δ

average client delay : Nδ/2

Algorithm efficient ?

Synchronization delay (loaded system)





average sync. delay : (N+1)δ/2

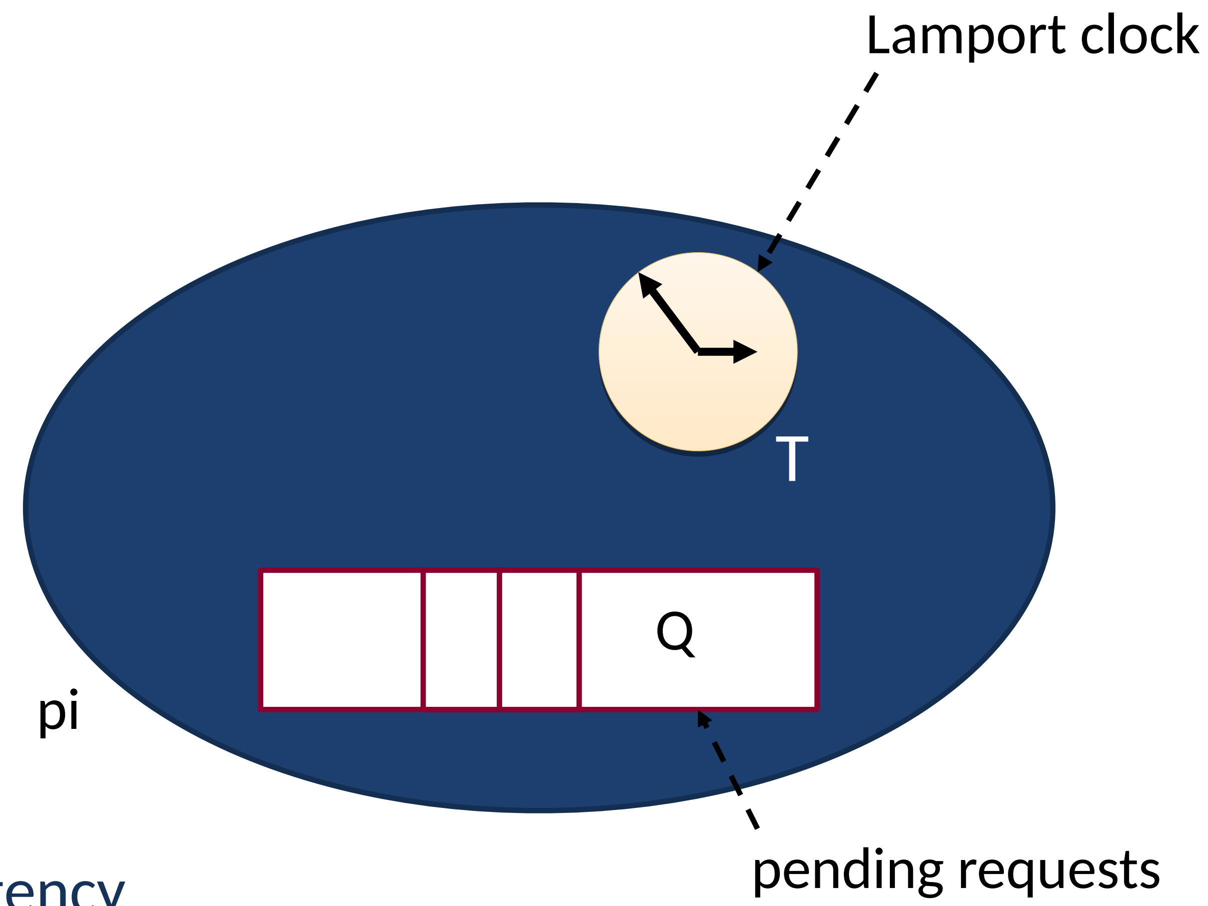
Multicast : Ricart-Agrawala algorithm

Filosophy

- Use multicast to reduce bandwidth
- Use logical clock (Lamport clock) to realize fairness
- Basic mechanism to enter CS :
  multicast request and enter if all others agree

Each process pi

- has **state** variable
  - *Released* : outside critical section
  - *Wanted* : wants to enter
  - *Held* : inside critical section
- Lamport clock **T**
- Queue **Q** to store pending Requests



Organization of Q ?

- Lamport-clock based happened-before
- total ordering implemented to ensure consistency

<p1,T1> **<** <p2,T2>
⇔ (T1<T2) or ((T1==T2) and (p1<p2))

Please see related file in Blackboard for more details on Lamport clocks

Ricart-Agrawala algorithm

Messages

- Request(pi,Ti),
- Reply

Each process pi

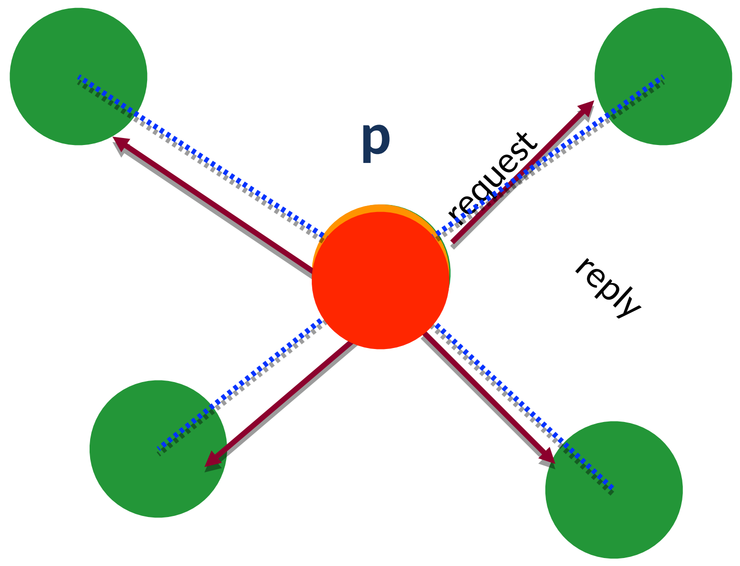
```
Initialization
                state = Released
enter()
                state = Wanted
                multicast Request(pi,Ti) to all processes
                store T=Ti of Request message
                wait until (N-1) Reply messages received
                state = Held
leave()
                state = Released
                send Reply to all pending Requests in Q

when receiving Request(pj,Tj) at pi (i≠j)
            if (state == Held) or ((state==Wanted) and (T,pi)<(Tj,pj)) {
                enqueue Request in Q
                // do NOT reply
            } else send Reply to pj
```

Examples

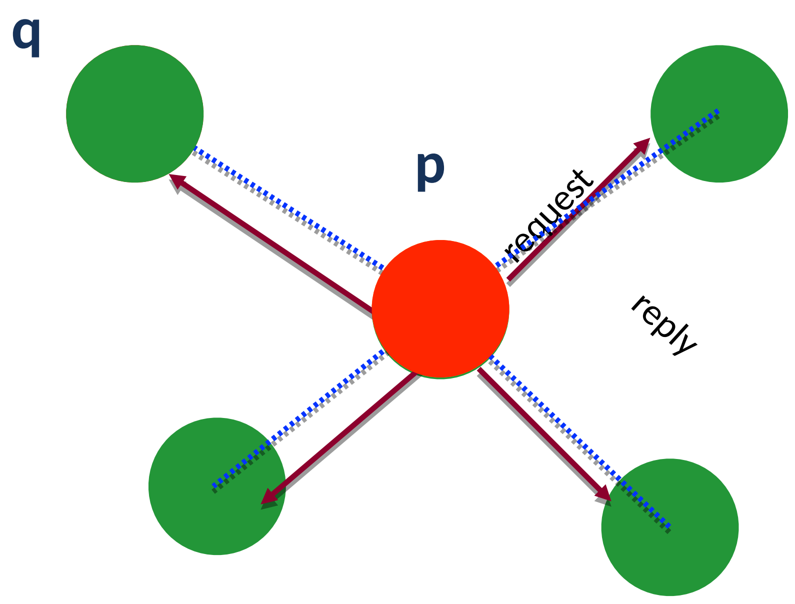
1. **Process p requests entry, all others in RELEASED-state**

   - all **(N-1)** other processes reply immediately
   - **p** can enter **CS**



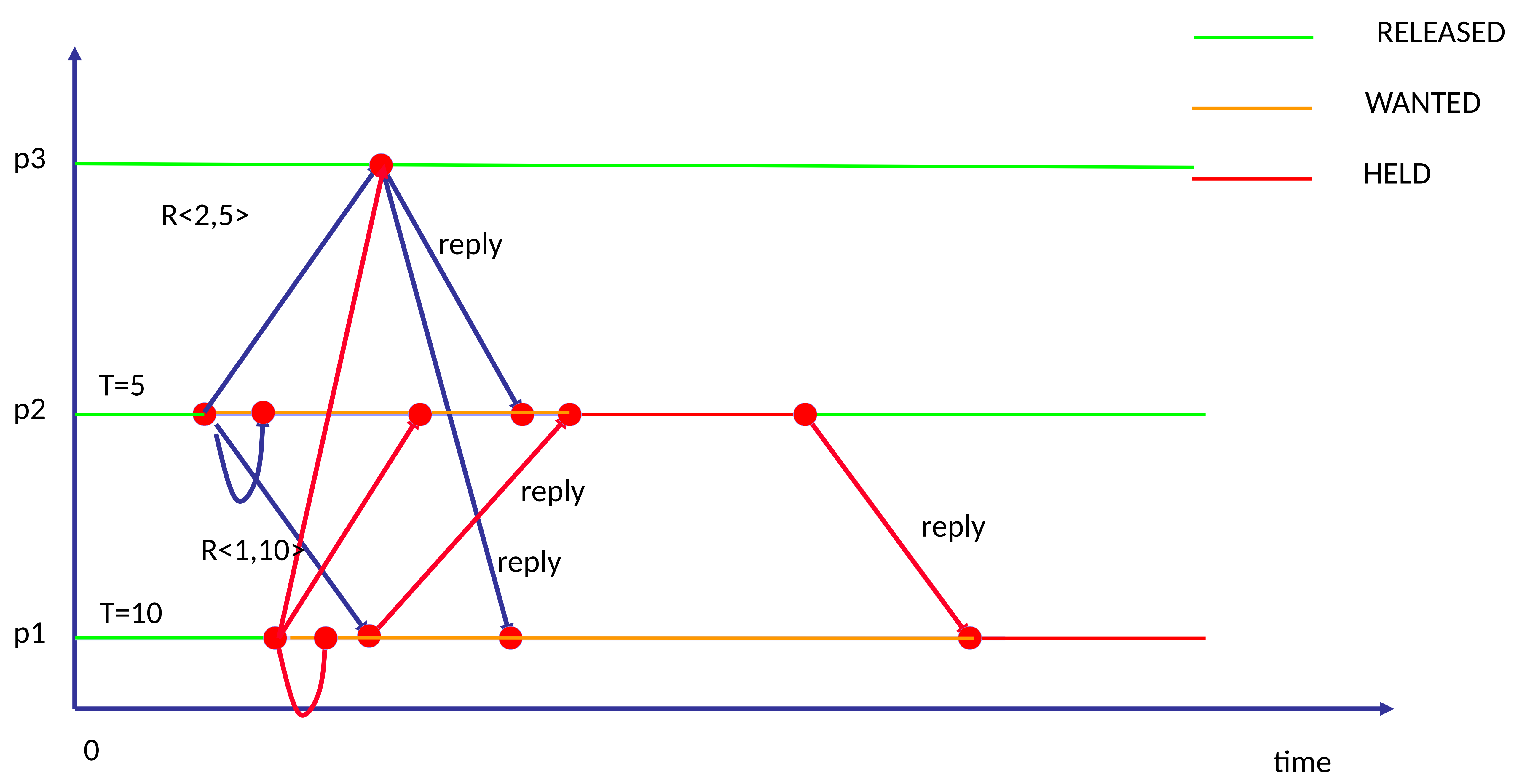
2. **Process p requests entry, one process q in HELD-state, all others in RELEASED-state**

   - **(N-2)** other processes reply immediately
   - **p** waits until q has left **CS**
   - **p** can enter **CS**



Examples

3. Processes p1 and p2 request entry, p3 not



Algorithm OK ?

1

Safety : a proof

Suppose **p** and **q** simultaneously executing critical section

- both **p** and **q** received **(N-1)** Reply-messages
- **p** sent **Reply** to **q** AND **q** sent **Reply** to **p**

- condition (state(i) == Held) or ((state(i)==Wanted) and (Ti,pi)<(Tj,pj))
  DOES NOT hold for
  (pj=q),(pi=p) (a)
  AND (pj=p),(pi=q) (b)

some logic for (a)

▯ !**[** (state(p) == Held) or ((state(p)==Wanted) and (Tp,p)<(Tq,q)) **]**

▯ (state(p)!=Held) AND **[** (state(p)!=Wanted) or (Tq,q)<(Tp,p) **]**

▯ **[** (state(p) != Held) and (state(p)!=Wanted) **]**
or **[** (state(p) != Held) and (Tq,q)<(Tp,p) **]**

▯ (state(p)==Released) or **[** (state(p) != Held) and (Tq,q)<(Tp,p) **]**

▯ **p** can NOT be in state **Released** (because now in **CS**)

▯ **[** (state(p) != Held) and (Tq,q)<(Tp,p) **]**

Algorithm OK ?

1

Safety : a proof

…
(a) => [(state(p) != Held) and (Tp,p)>(Tq,q)]
(b) => [(state(q) != Held) and (Tq,q)>(Tp,p)]

CAN NOT hold simultaneously
=> **p** and **q** can NOT be executing simultaneously in critical section

2

Liveness

Every Request eventually granted
- immediately
- after dequeueing from **Q**

⇒ every process will eventually receive **(N-1)** answers

3

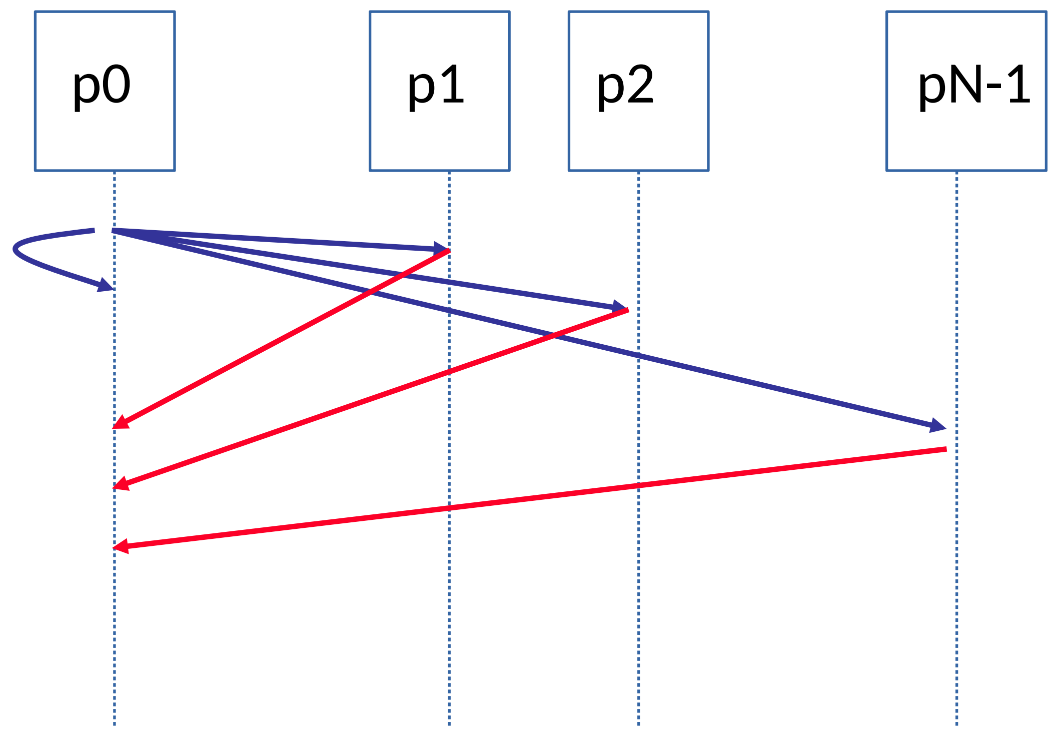
Fairness

Requests replied in happened-before order

Algorithm efficient ?

Bandwidth usage

enter()



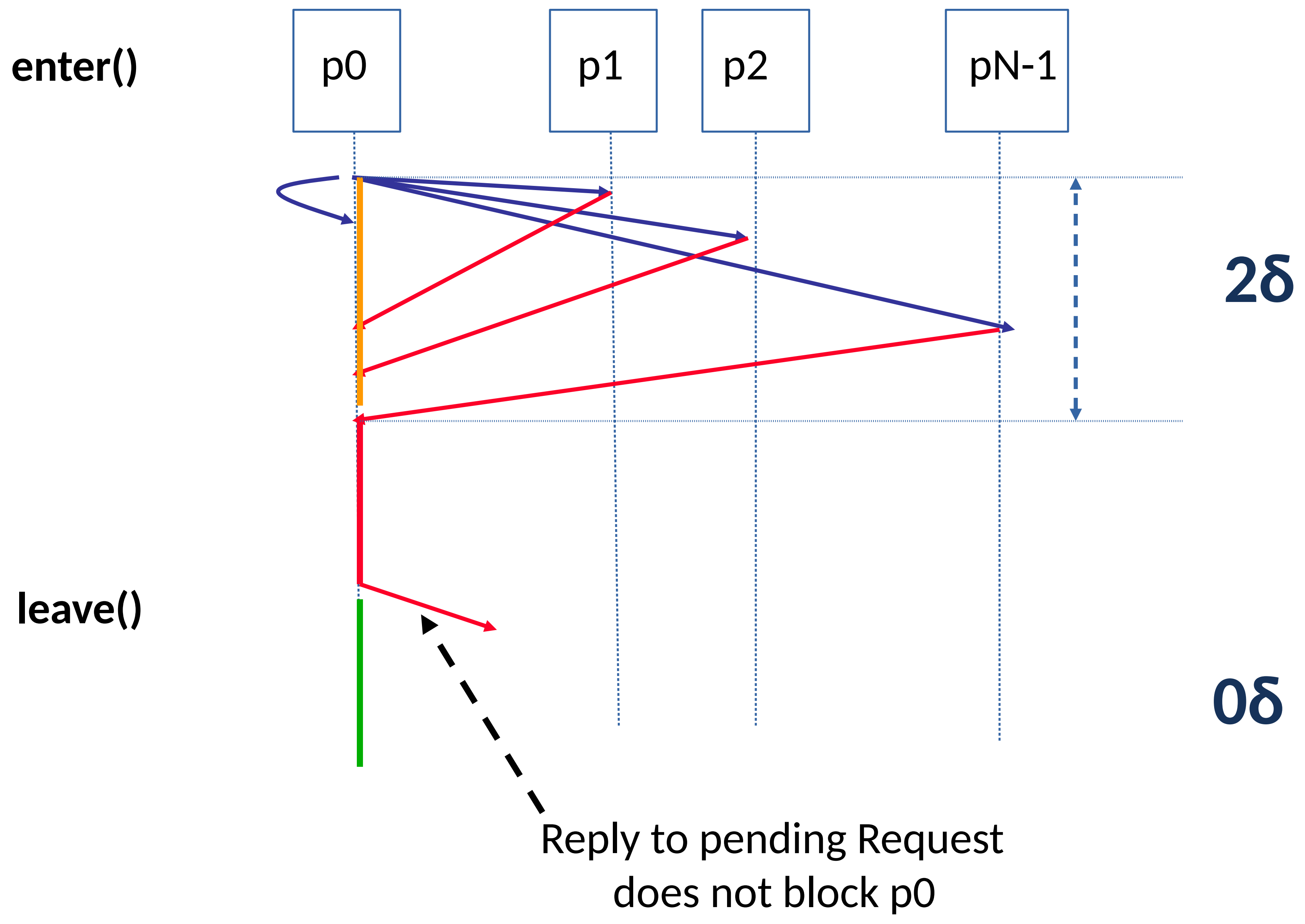
N Request messages
(N-1) Reply messages

if multicast supported
  1 Request message only !

leave() no messages needed for leaving

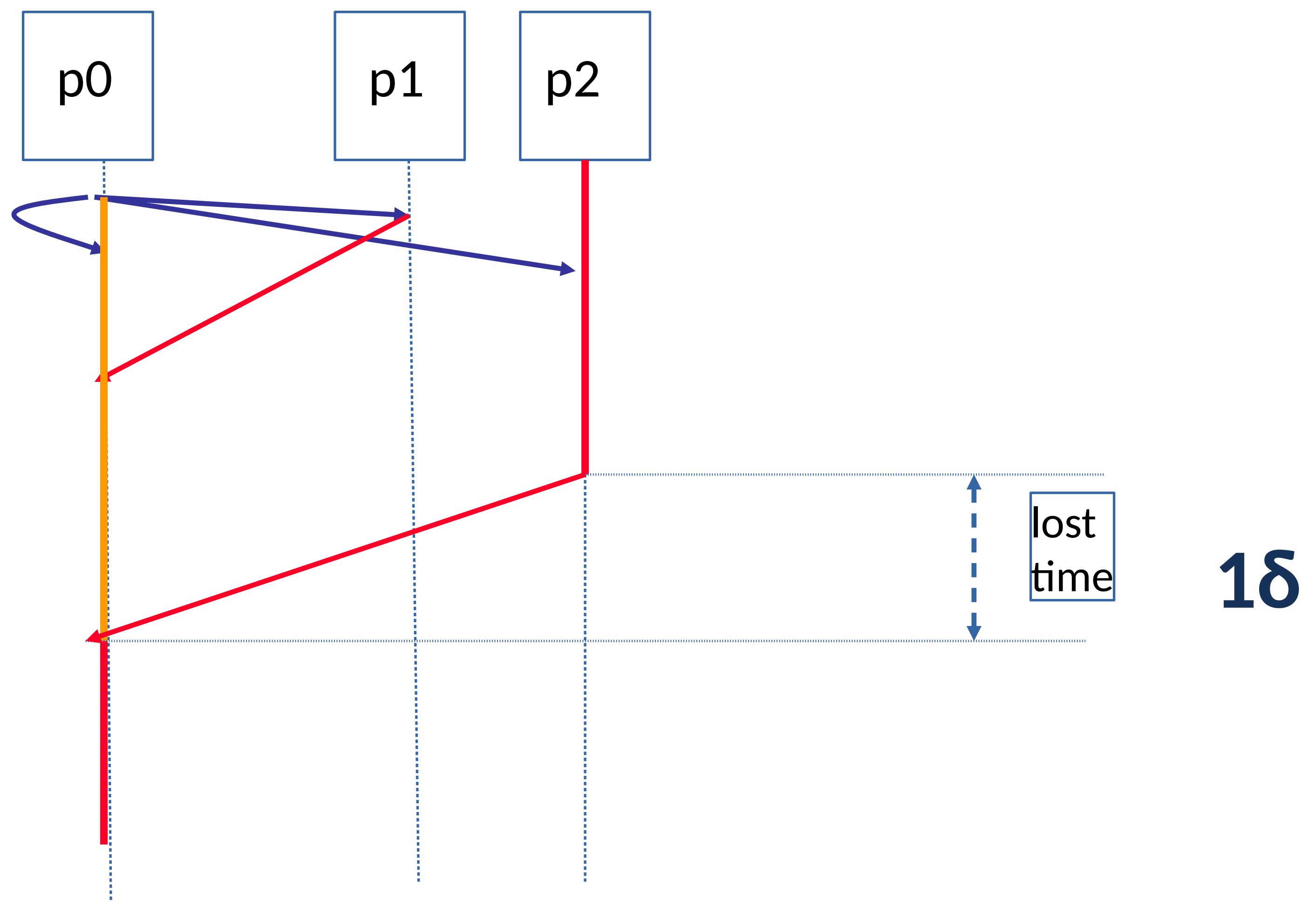
Algorithm efficient ?

Client delay



Algorithm efficient ?

Synchronization delay (loaded system)

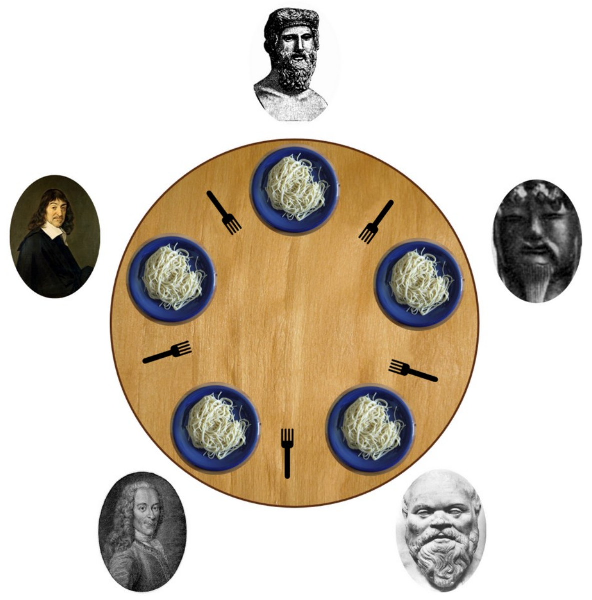


Summary on efficiency

	Bandwidth enter()	Bandwidth leave()	Client delay enter()	Client delay leave()	Synchronization delay
Central server	2M	1M	2δ	0δ	2δ
Ring algorithm	constant (N+1)/2		Nδ/2		(N+1)δ/2
Ricart-Agrawala	(2N-1)M	0M	2δ	0δ	1δ

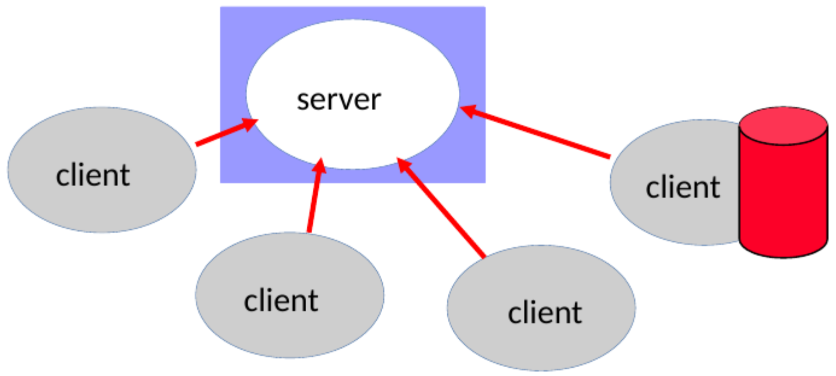
N: number of processes
M: number of voting sets that each process belongs to
δ : cost of sending a message

Summarizing



Coordination

- Basic Operations
- Desirable Properties:
  - Safety, Liveness, Fairness
- Evaluation Metrics:
  - Bandwidth, Client delay, Synchronization delay



Discussed Algorithms

- Central server algorithm
- Ring Algorithm
- Ricart-Agrawala Algorithm

Questions?

Coordination

Part-2