Coordination

Part-2

Distributed Mutual Exclusion

Election Mechanisms

Maekawa Voting

Filosophy

- Drawback of Ricart-Agrawala : all processes need to agree —> linear scaling
- Maekawa voting : only SUBSET of processes involved
- Basic idea : “vote on behalf of others”
- Candidate process must collect sufficient votes before entering

Voting set for each process pi : Vi

- $V_i \subseteq \{p_1, \ldots, p_N\}$ (Required)
- $p_i \in V_i$ (Required)
- $\boxed{V_i \cap V_j \neq}$ (Required)
- $|V_i| = K$ (fairness) (Bonus)
- Every pj contained in exactly M voting sets (Bonus)

Basic idea

- process $q \in (V_i \cap V_j)$
- —> q makes sure pi and p_j are not simultaneously executing critical section
- —> safety condition met
- q only votes for one process
- Additional state needed per process : **voted**

The algorithm

Process i requesting access

Initialization

    state = Released
    voted = False

enter()

    state = Wanted
    multicast **Request(pi)** to voting set **Vi**
    wait until **K** Reply messages received
    state = Held

leave()

    state = Released
    multicast **Release** to voting set **Vi**

Member of Vi receiving request

when receiving Request(pj) at pi

```
if (state == Held) or (voted == True) {
        enqueue Request in Q
        // do NOT reply
} else {
    send Reply to pj
    voted = True
}
```

when receiving Release(pj) at pi

```
if (Q != empty) {
    dequeue pending Request m from Q
    send Reply to sender(m)
    voted = True
} else {
    voted = False
}
```

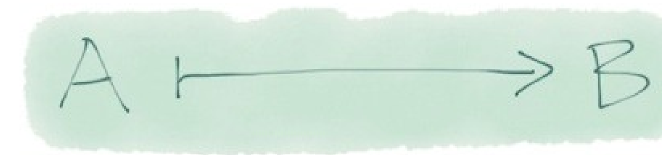
Differences w.r.t. Ricart-Agrawala

- additional state variable per process needed (voted or not)
- multicast request to enter to voting set only
- explicit leave needed (so voting processes can vote for other process)

Constructing voting sets

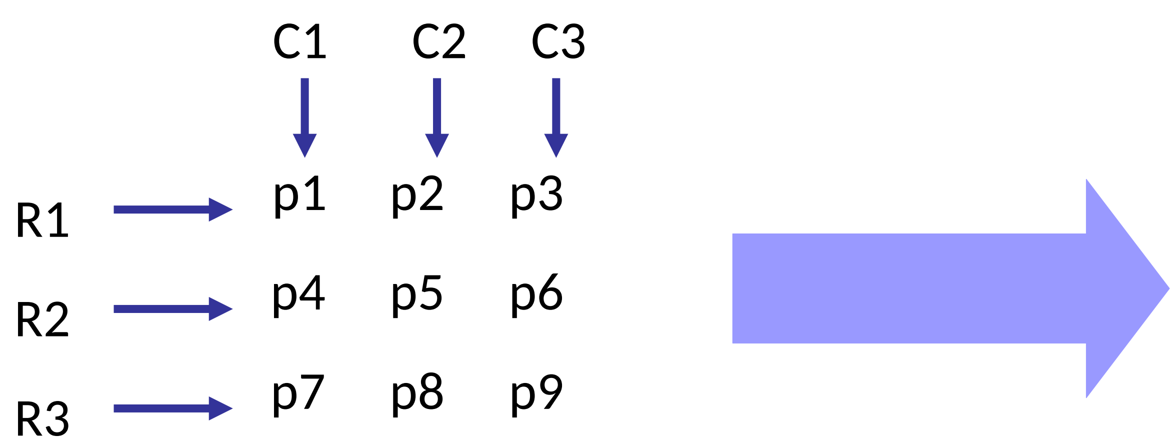
Choosing parameters

- Theoretical result : optimal solution (minimal K)
  - K ≈ sqrt(N)
  - M = K
- In practice : difficult to calculate optimal Vi
  Sub-optimal solution
  - K ≈ 2sqrt(N)
  - M = K

Theory: 

Practical algorithm (for N=S²)

for S=3



V1 = R1 ∪ C1
V2 = R1 ∪ C2
V3 = R1 ∪ C3
V4 = R2 ∪ C1
V5 = R2 ∪ C2
V6 = R2 ∪ C3
V7 = R3 ∪ C1
V8 = R3 ∪ C2
V9 = R3 ∪ C3

Constructing voting sets

In general

Construct **SxS** matrix **A**, consisting of all processes

- **i** is row where **p** is found
- **j** is column where **p** is found
- **Ri** = **i**-th row of **A**
- **Cj** = **j**-th column of **A**

Voting set **Vp** = **Ri** ∪ **Cj**

$$
\begin{array}{cccc}
 & C1 & C2 & C3 \\
 & \downarrow & \downarrow & \downarrow \\
R1 \rightarrow & p1 & p2 & p3 \\
R2 \rightarrow & p4 & p5 & p6 \\
R3 \rightarrow & p7 & p8 & p9
\end{array}
$$

Voting set for each process pi : Vi

- $V_i \subseteq \{p_1, \ldots, p_N\}$, satisfying
- $p_i \in V_i$
- $V_i \cap V_j \neq$
- $|V_i| = K$ (fairness)
- pj contained in M voting sets

Checking this Vp ...

$$p \in Ri,\ p \in Cj \;\Rightarrow\; p \in Vp$$

$$Vp = Ri \cup Cj$$
$$Vq = Rs \cup Ct$$
$$\Rightarrow Vp \cap Vq = (Ri \cup Cj) \cap (Rs \cup Ct)$$
$$= (Ri \cap Rs) \cup (Ri \cap Ct) \cup (Cj \cap Rs) \cup (Cj \cap Ct)$$

BUT $Rk \cap Cl \neq \Phi$ (for any k,l) ($skl \in Rk$ and $skl \in Cl$)

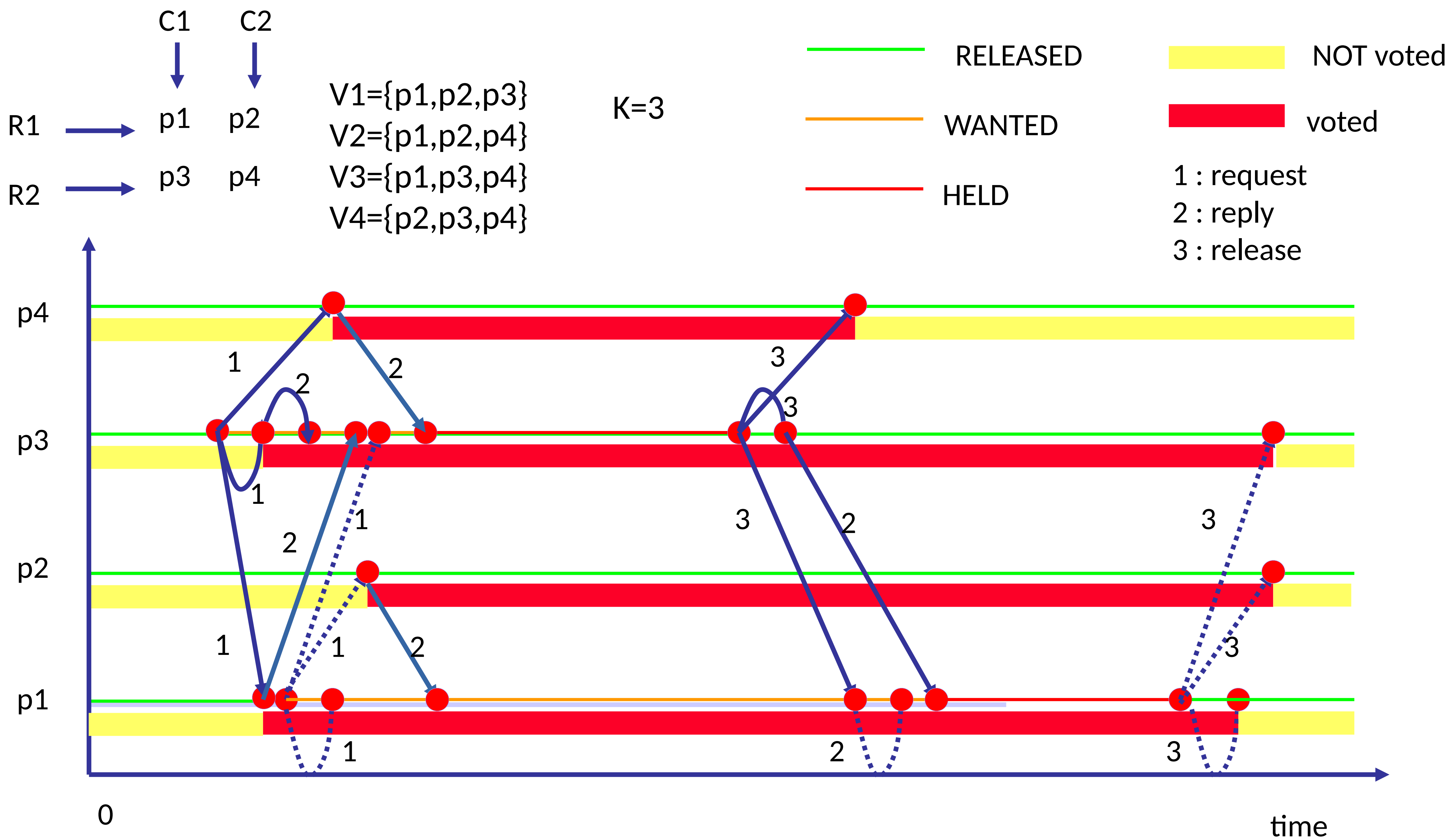
$\Rightarrow Vp \cap Vq \neq \Phi$

$K = 2S - 1$

$K = M$

Example

Processes p1 and p3 request entry, p2 and p4 not



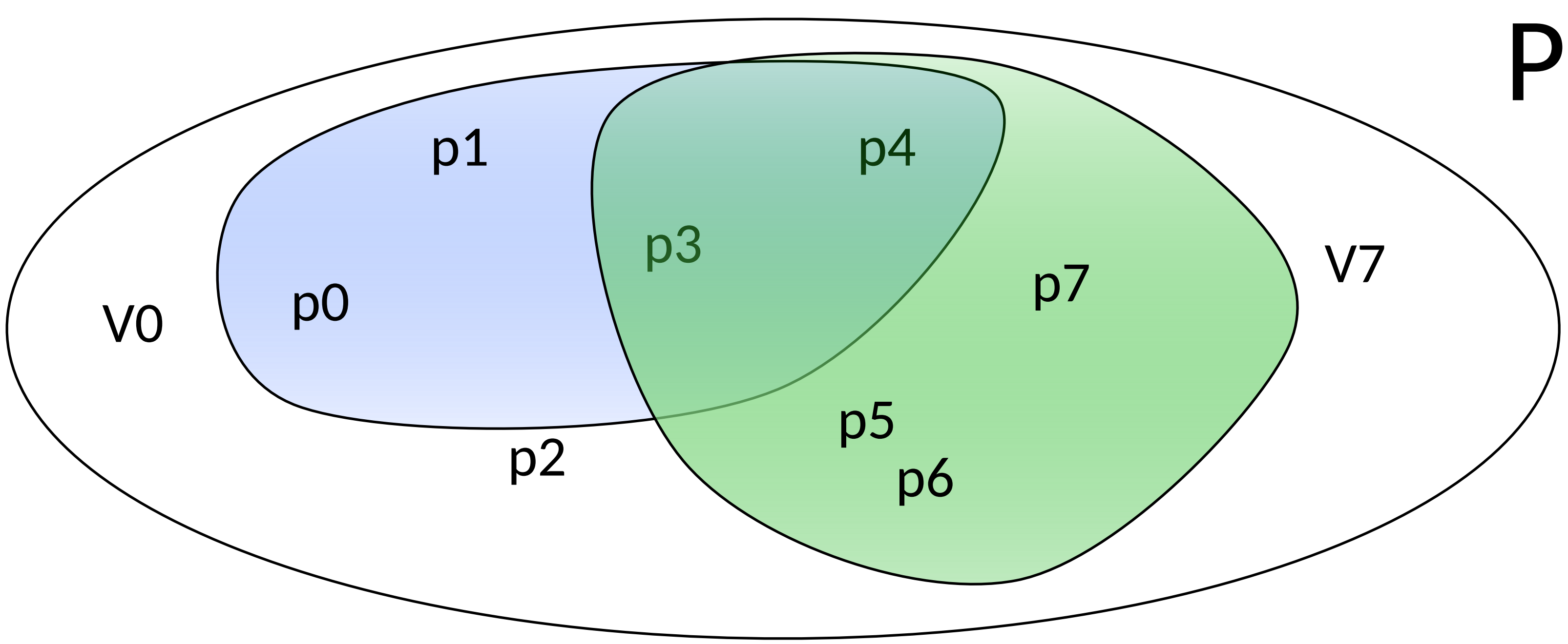
Algorithm OK ?

1 Safety

Suppose **p** and **q** simultaneously active
=> all processes in **Vp** and **Vq** voted for **p** AND **q**
=> process **t** in **Vp** ∩ **Vq** ≠ **Φ** voted for **p** AND **q**
impossible :
    voted=True immediately after voting for 1 process

2 Liveness

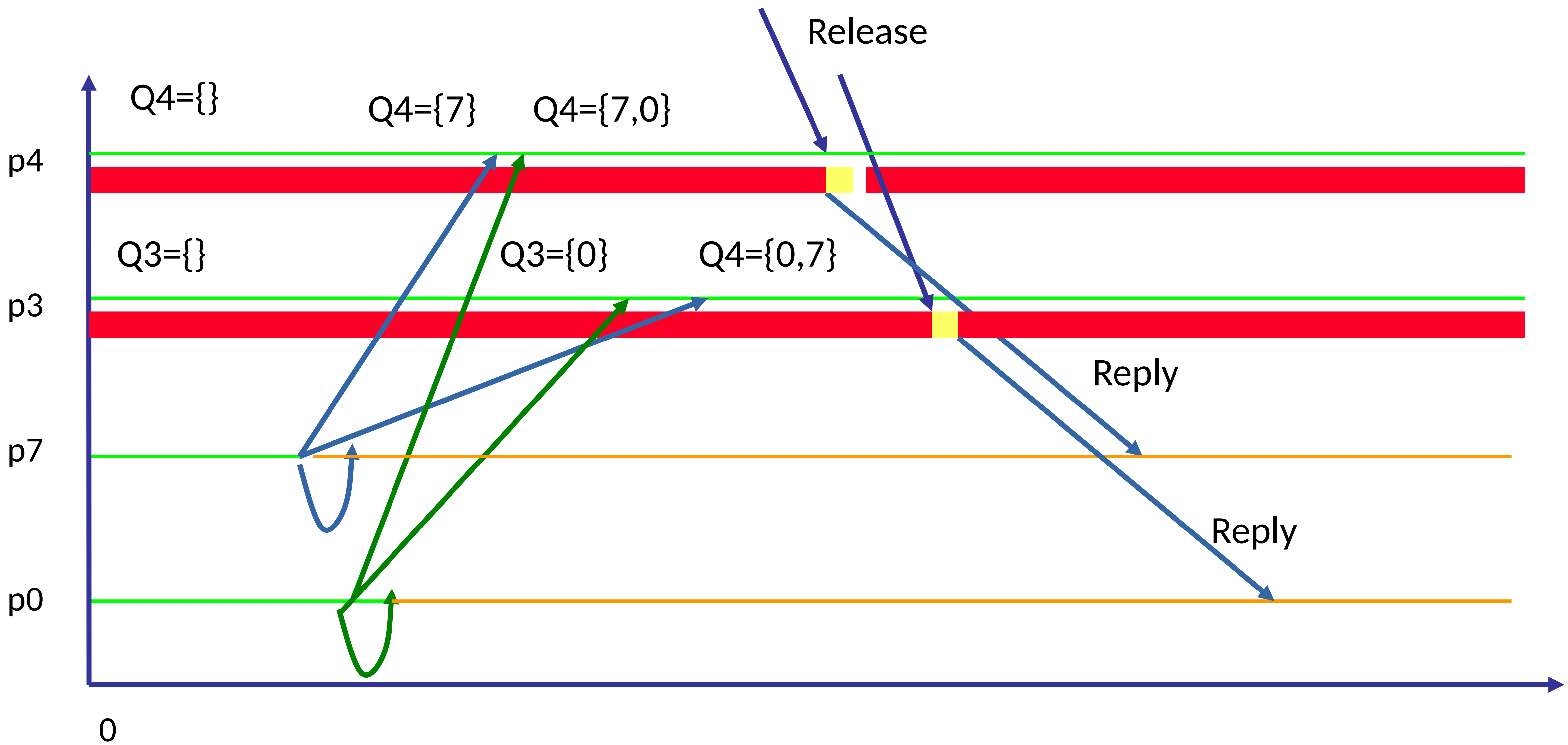
deadlock prone !!!



V0 ∩ V7 = {p3,p4}

should vote consistently !!!

Algorithm NOT OK !



p7 and p0 will NEVER receive Reply from complete voting set !

Algorithm OK ?

1 Safety

Suppose p and q simultaneously active
=> all processes in Vp and Vq voted for p AND q
=> process t in $Vp \cap Vq \neq \Phi$ voted for p AND q
impossible :
voted=True immediately after voting for 1 process

2 Liveness

deadlock prone !!!
Solution: use totally ordered Lamport clocks

3 Fairness

Guaranteed if we use Lamport clocks

Algorithm efficient ?

Bandwidth usage

enter()

same as Ricart-Agrawala, but message sent to voting set only !

-> **K Request** messages
-> **K Reply** messages

leave()

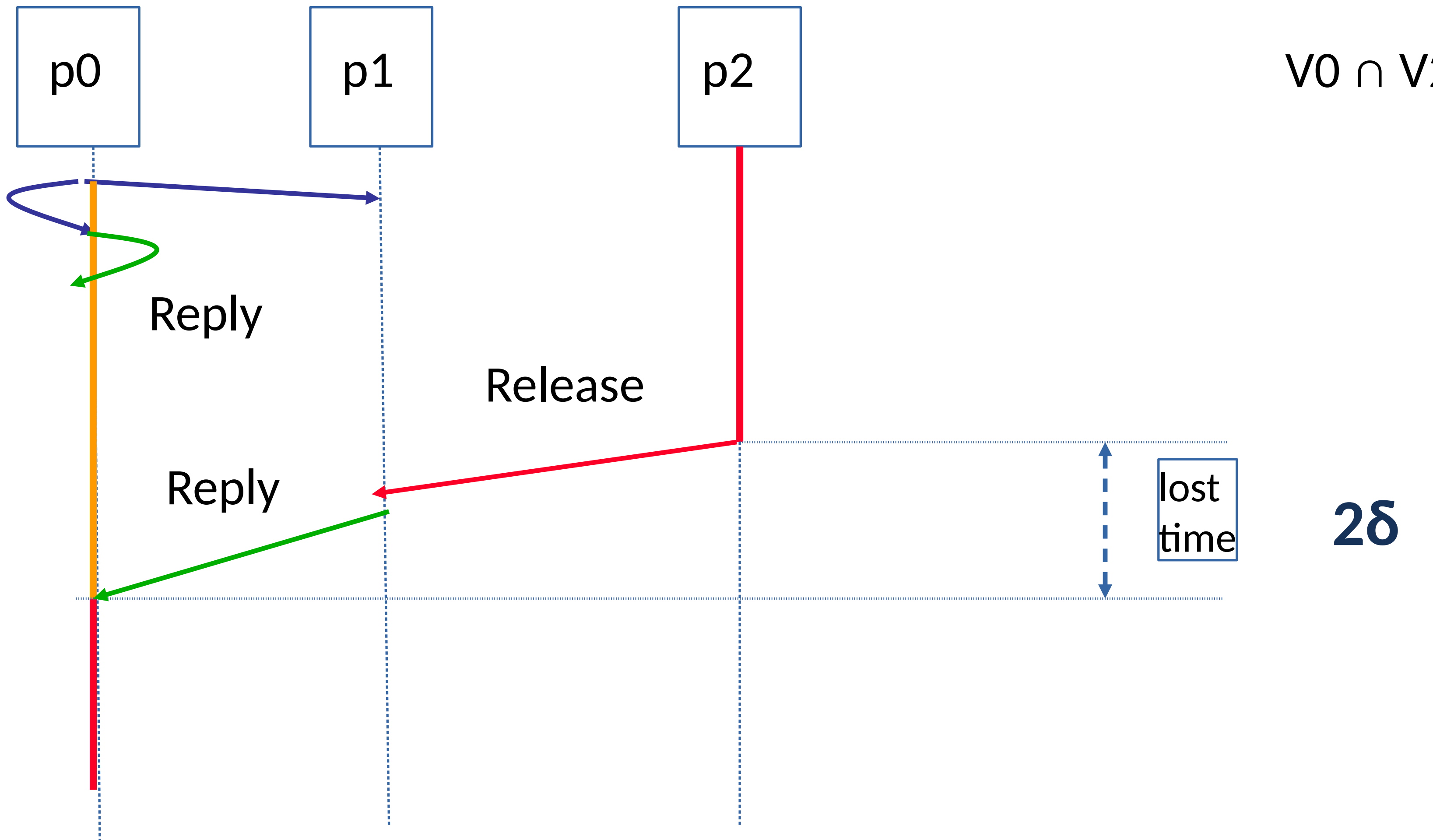
explicit **Leave** message now needed
-> **K Release** messages

Client delay

same as Ricart-Agrawala
enter() : 2δ
leave() : 0δ

Algorithm efficient ?

Synchronization delay



V0 ∩ V2={p1}

Summary on efficiency

	Bandwidth		Client delay		Synchronization delay
	enter()	leave()	enter()	leave()	
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
Maekawa voting	2KM	KM	2δ	0δ	2δ

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

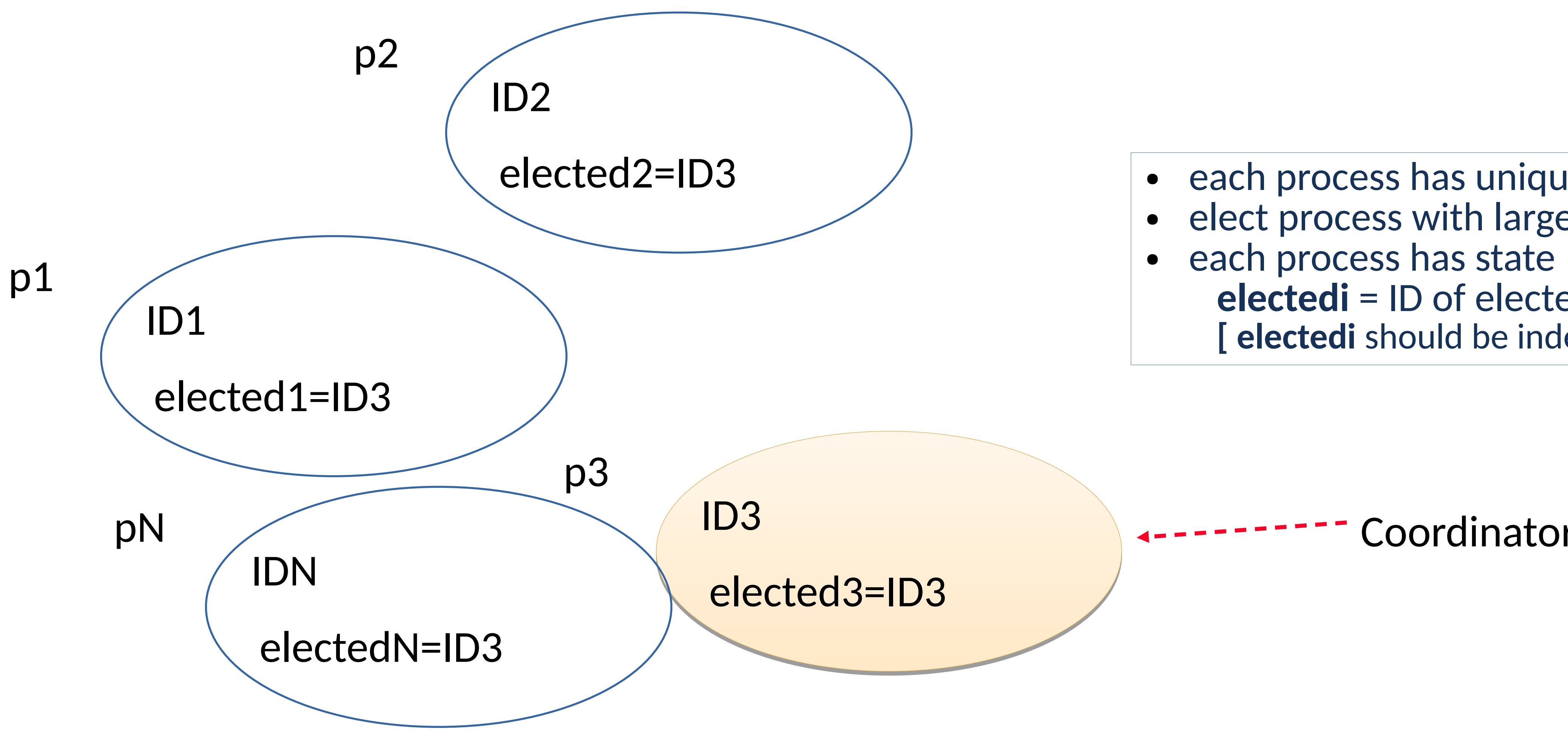
Distributed
Mutual Exclusion

Election
Mechanisms

The Election Problem

Consider

- **N** processes **{p1, … , pN}**
  - NO shared variables
  - knowing each other (can communicate)
- select ONE process to play special role (e.g. coordinator)
- every process **pi** should have same coordinator
- if elected process fails : do new election round

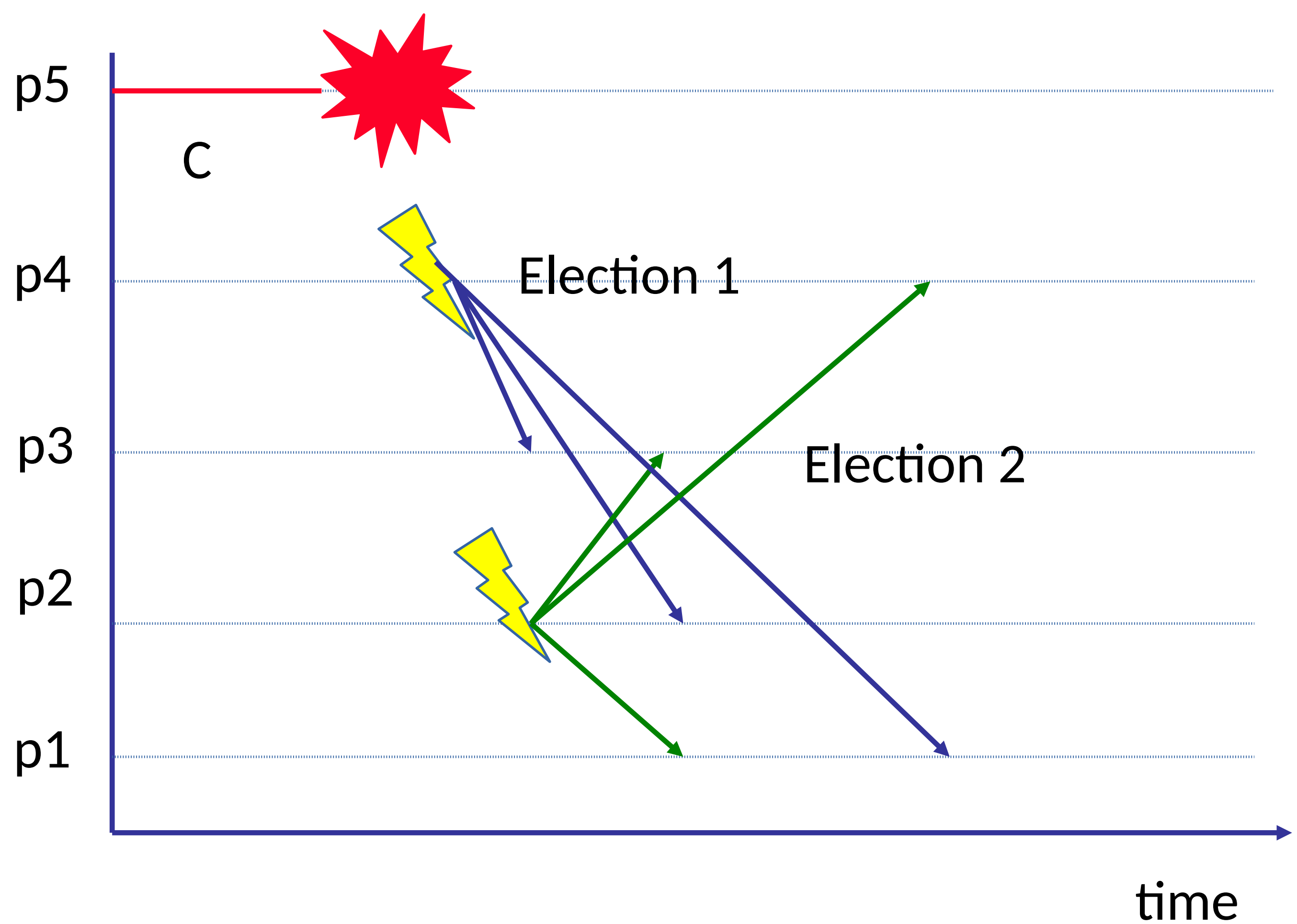


- each process has unique ID
- elect process with largest ID
- each process has state
  electedi = ID of elected process
  [**electedi** should be independent of **i**]

Some terms

Assumed environment

- request election : a process "calls the election"
  at most 1 election initiated per process
  possibly **N** elections running simultaneously
- at any time, a process is either
  - *participant* : currently engaged in some election
  - *non-participant* : currently not engaged in any election



Good elections

Correctness requirements

1 **Safety (REQUIRED)**

each participant process **pi** has :

electedi = ?

OR **electedi** = **P**

(**P** is elected process, non-crashed with largest ID)

2 **Liveness (REQUIRED)**

all processes **pi** participate

eventually set **electedi** ≠ ? or crash

Evaluation metrics

- **Bandwidth**

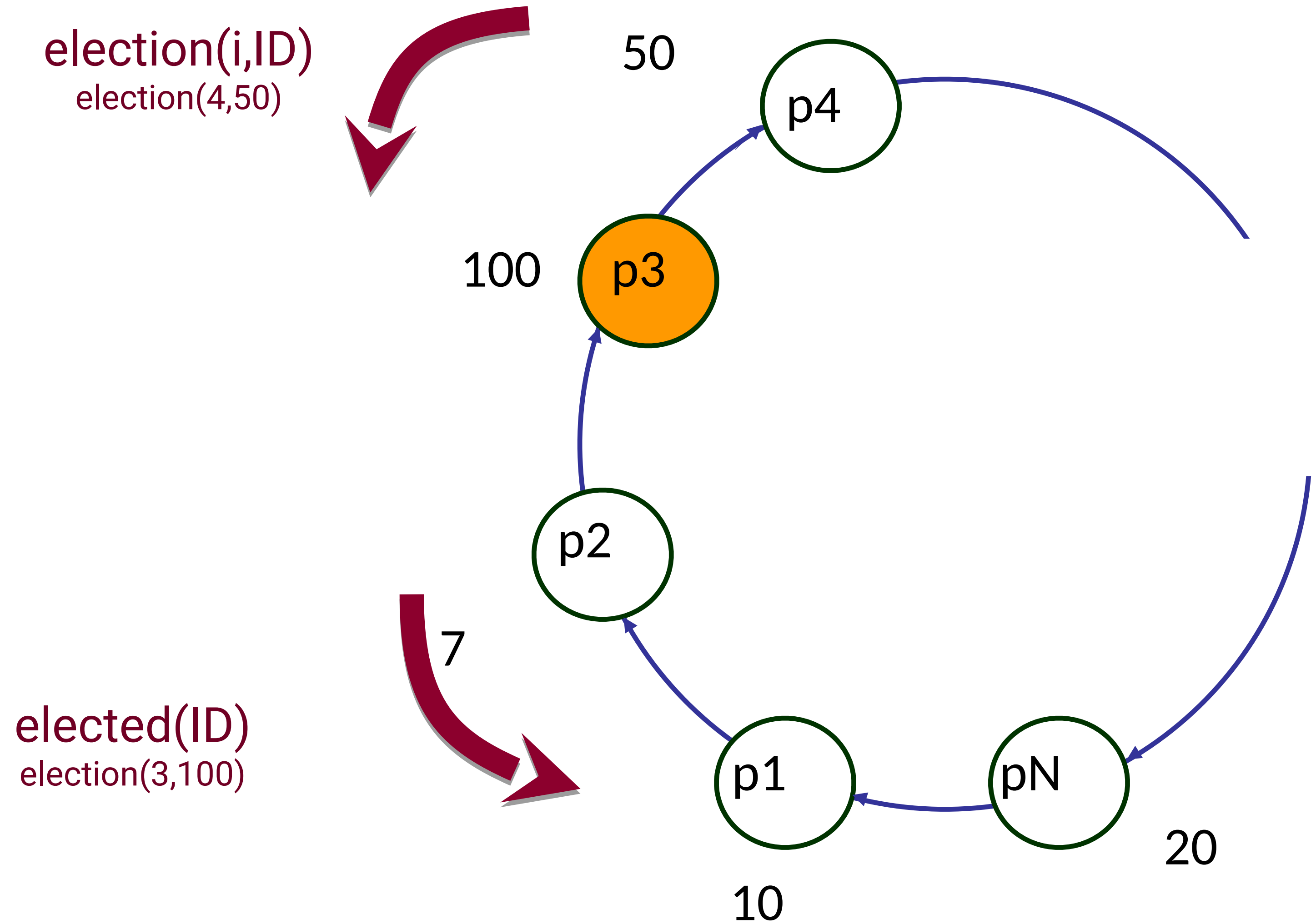
  # messages needed to do election process

- **turnaround time**

  time needed for election round

Ring algorithm: Chang – Roberts

- processes organized in ring
- non-identical IDs (how to make IDs unique ?)
- processes know how to communicate



Failure modes

No failures :

- reliable channels
- no process crashes

Asynchronous system

Algorithm

Each process p

Initialization

participant$_i$ = FALSE   for all i

Start election process p_i

participant$_i$ = TRUE
send message Election(i,ID$_i$)

Receipt of Elected(i)-message at p_j

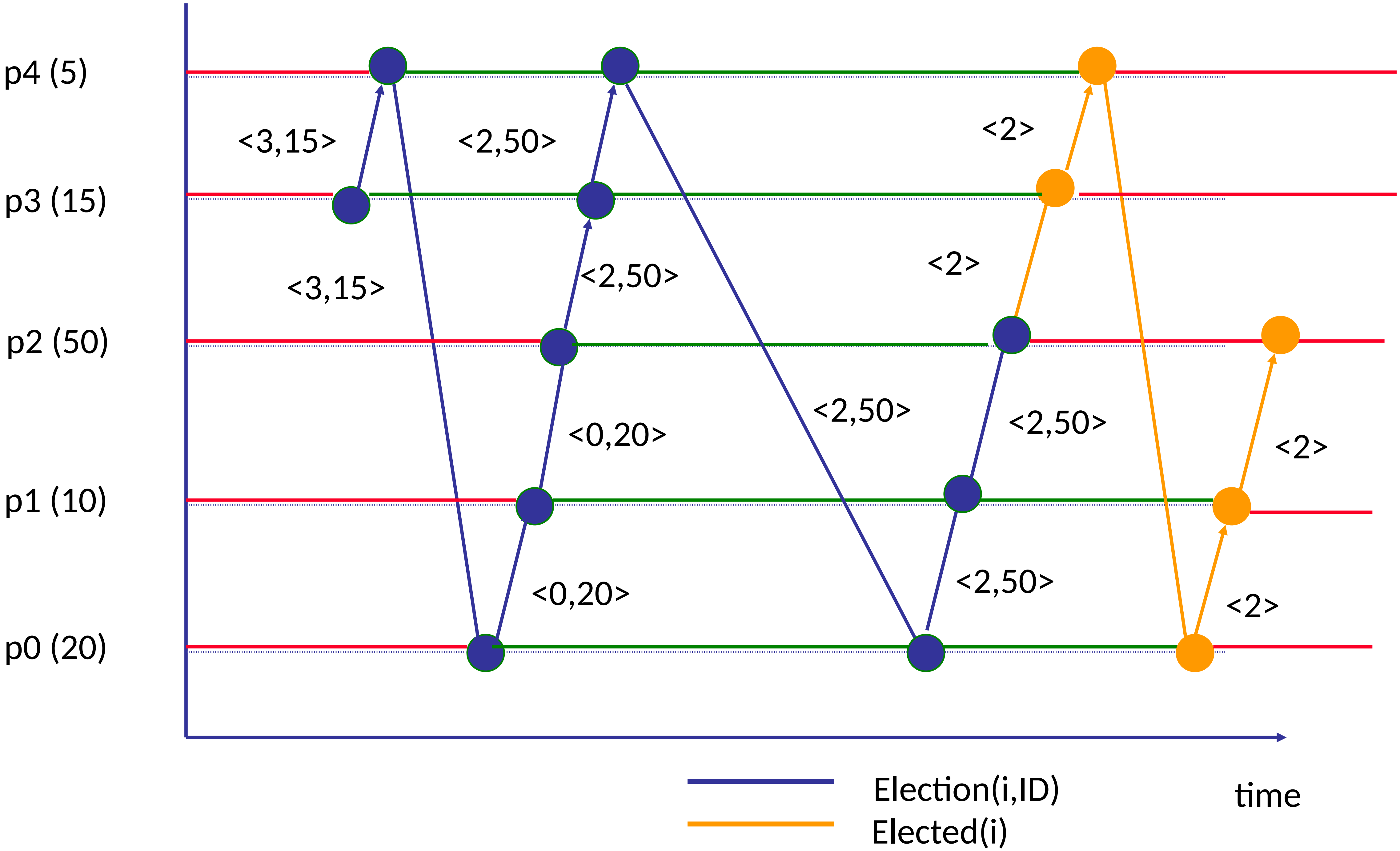
```
if(i≠j) {
    participantj = FALSE
    electedj = i
    forward Elected(i)

}
```

Receipt of Election(i,ID)-message at p_j

```
if(ID>IDj) {
    forward Election(i,ID)
    participant_j = TRUE
}
if((ID≤ID_j)and(i≠j)) {
    if(participant_j == FALSE) {
        send Election(j,ID_j)
        participant_j=TRUE
    }
}
if(i==j) {
    participant_j=FALSE
    elected_j = j
    send Elected(j)
}
```

Example

p3 calls the election



Algorithm OK ?

1 **Safety**

Elected message only sent if Election-message with own ID received
Suppose **p** and **q** both elected
=> **p** received **Elected(p)**
q received **Elected(q)**
BUT ID's are unique
(IDp<IDq) => **q** will NOT forward **Elected(p,IDp)**
(IDp>IDq) => **p** will NOT forward **Elected(q,IDq)**
=> impossible for BOTH messages to visit complete ring
=> impossible **p** AND **q** to be elected

2 **Liveness**

No failures
=> messages allowed to circulate
=> circulation stops (through participant state variable)

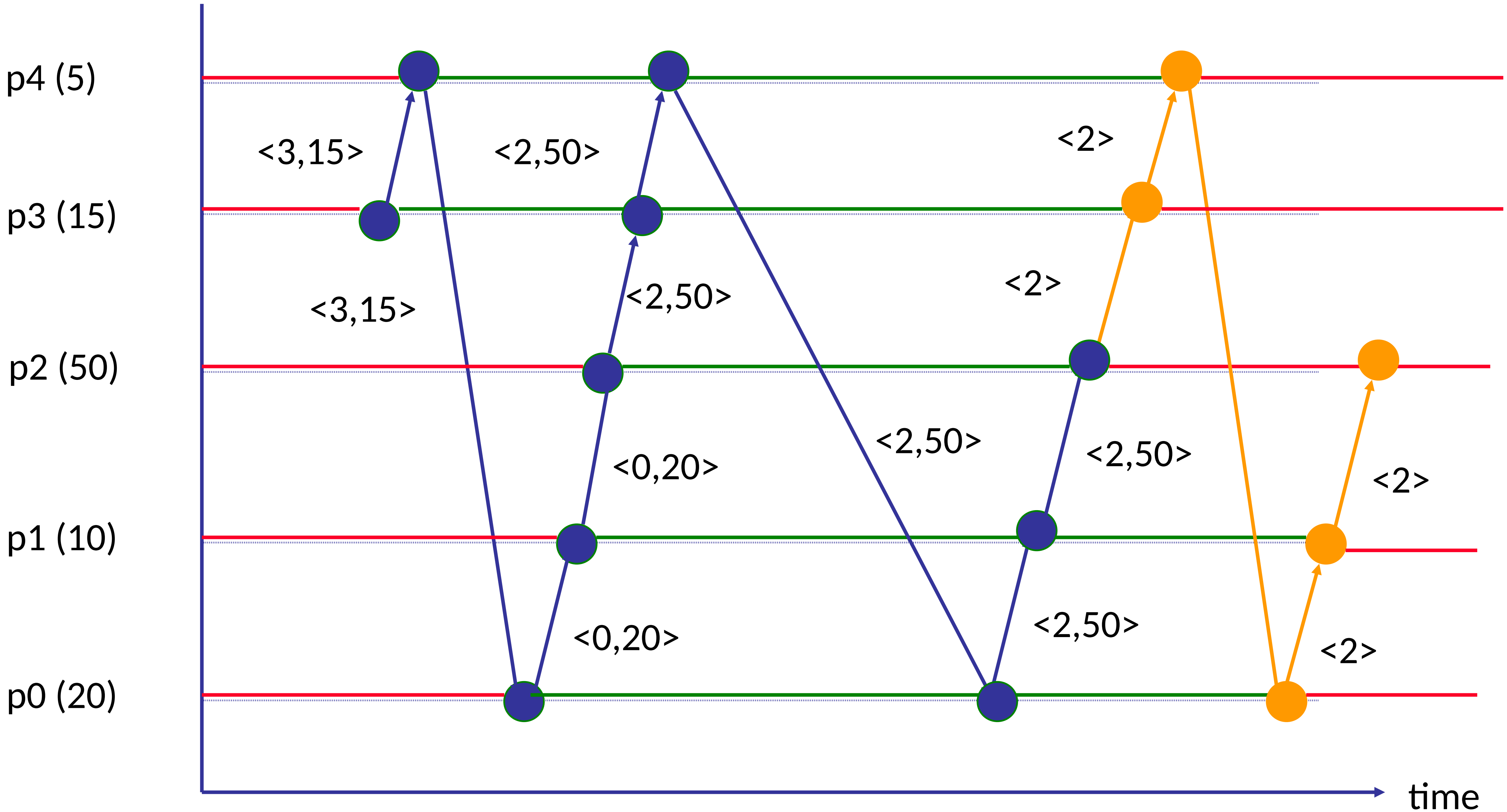
Algorithm efficient ?

3 phases

1. ID in Election message grows
2. do complete round with constant ID
3. let the Elected message circulate

Worst case : process with max ID is last process visited

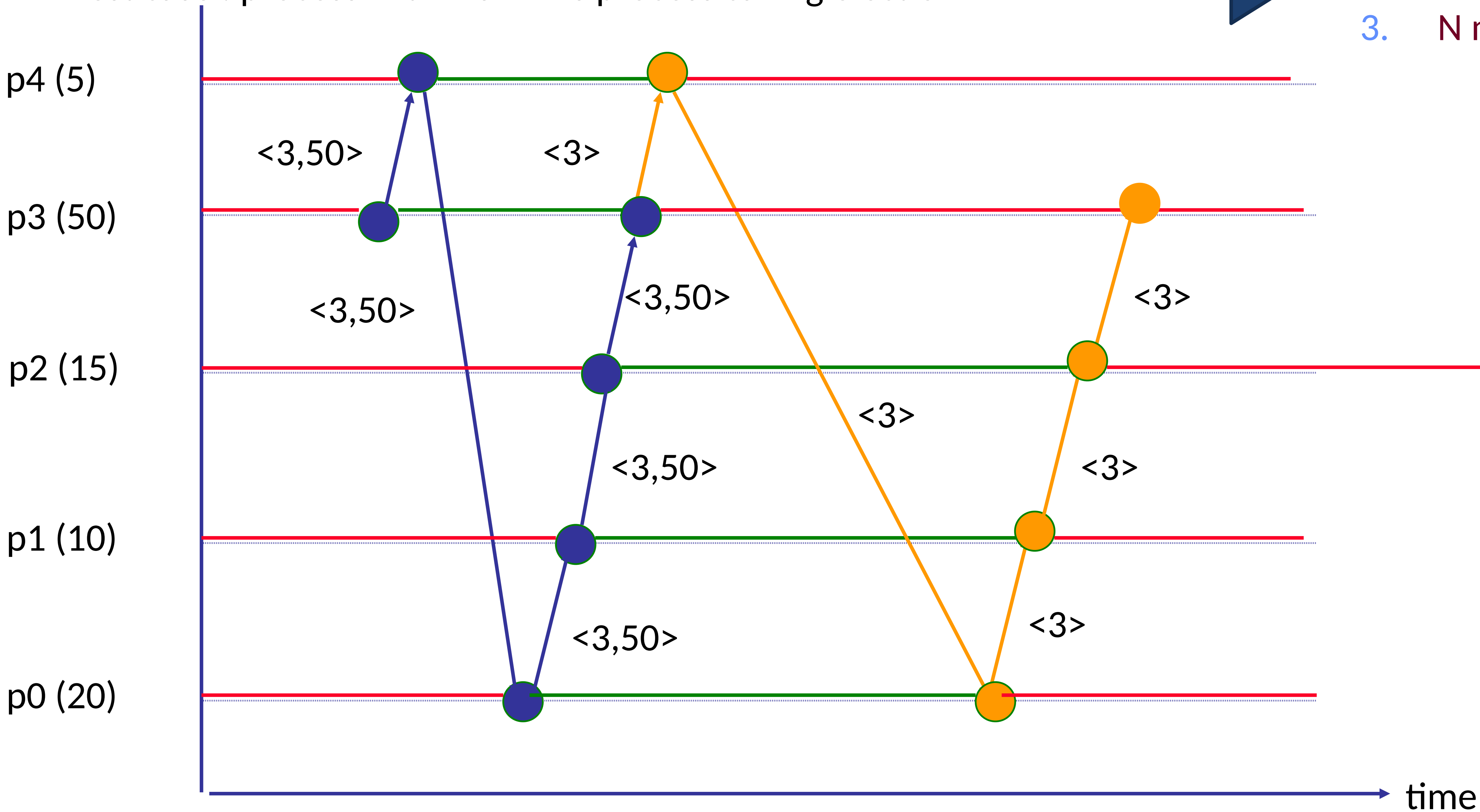
1. N messages
2. (N-1) messages
3. N messages



Algorithm efficient ?

Best case : process with max ID is process calling election

1. 1 message
2. (N-1) messages
3. N messages



	min	**max**	**average**
bandwidth	2NM	(3N-1)M	(5N-1)M/2
turnaround time	2Nδ	(3N-1)δ	(5N-1)δ/2

Bully algorithm (Garcia – Molina)

Context

- Failure mode:process crashes dealt with
- System model: Synchronous system (uses time-outs to detect failure)
- A-priori knowledge: process knows all processes with larger ID

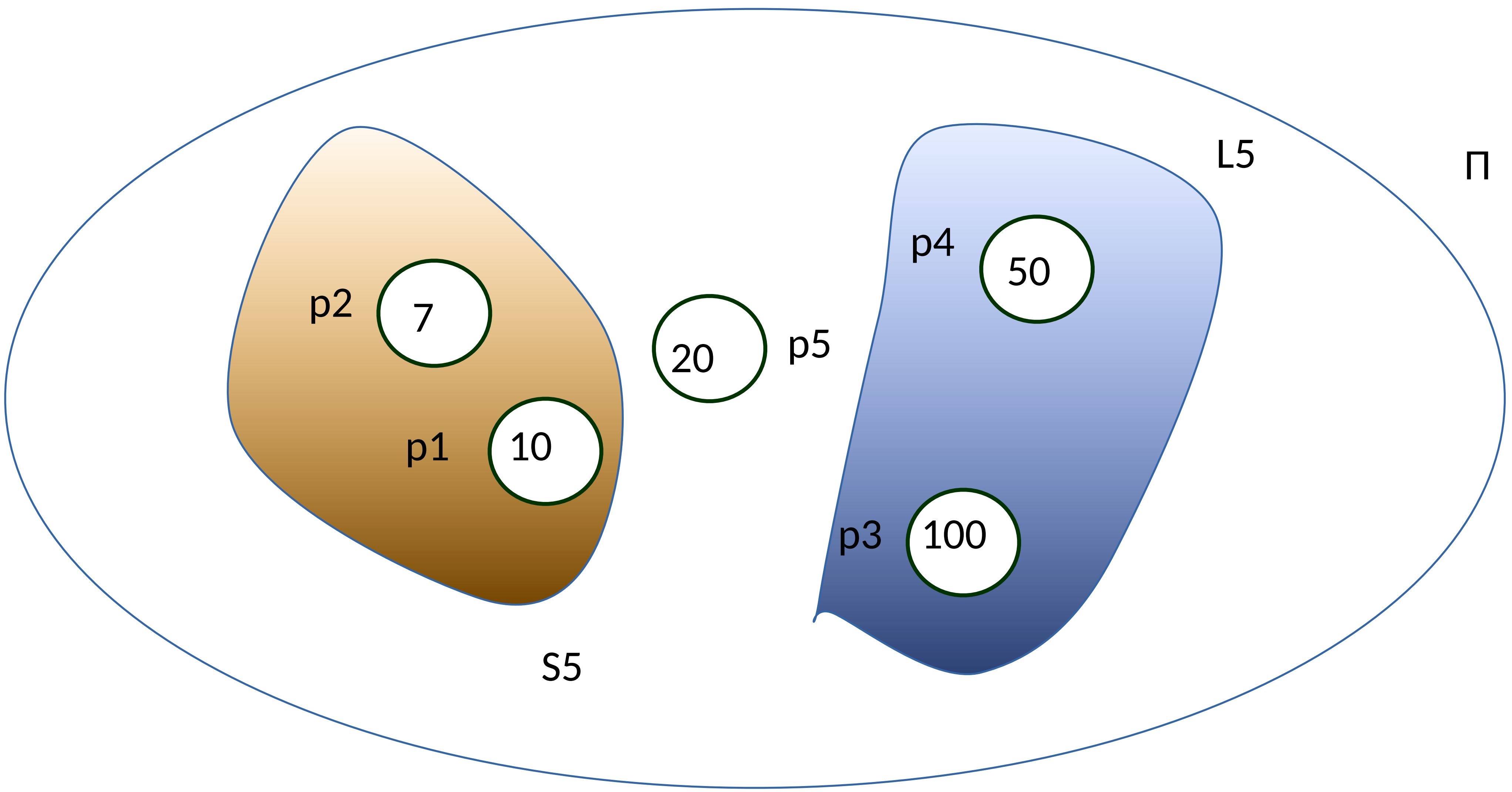
Filosophy

- Election starts when current coordinator fails
- Failure discovery :
  - by timeouts
  - election possibly by several processes
- Each process has
  - set **L** of candidate coordinators (set of processes with larger ID)
  - set **S** of other processes (smaller IDs)
- Upper bound for answering : **T**

Communication

Messages involved

- **election** announce election round
- **answer** reply to election message
- **coordinator** announce ID of elected coordinator



Algorithm

Call the election process p_i

if {**L**==   } then {
elected=i
send **coordinator(i)** to **S**
} else {
send **election(i)** to all processes in **L**
if no **answer-message** in period **T** then {
elected=i;
send **coordinator(i)** to **S**
} else {
if no **coordinator-message** in **T'** then call election again
}
}

Receipt coordinator(j) at pi

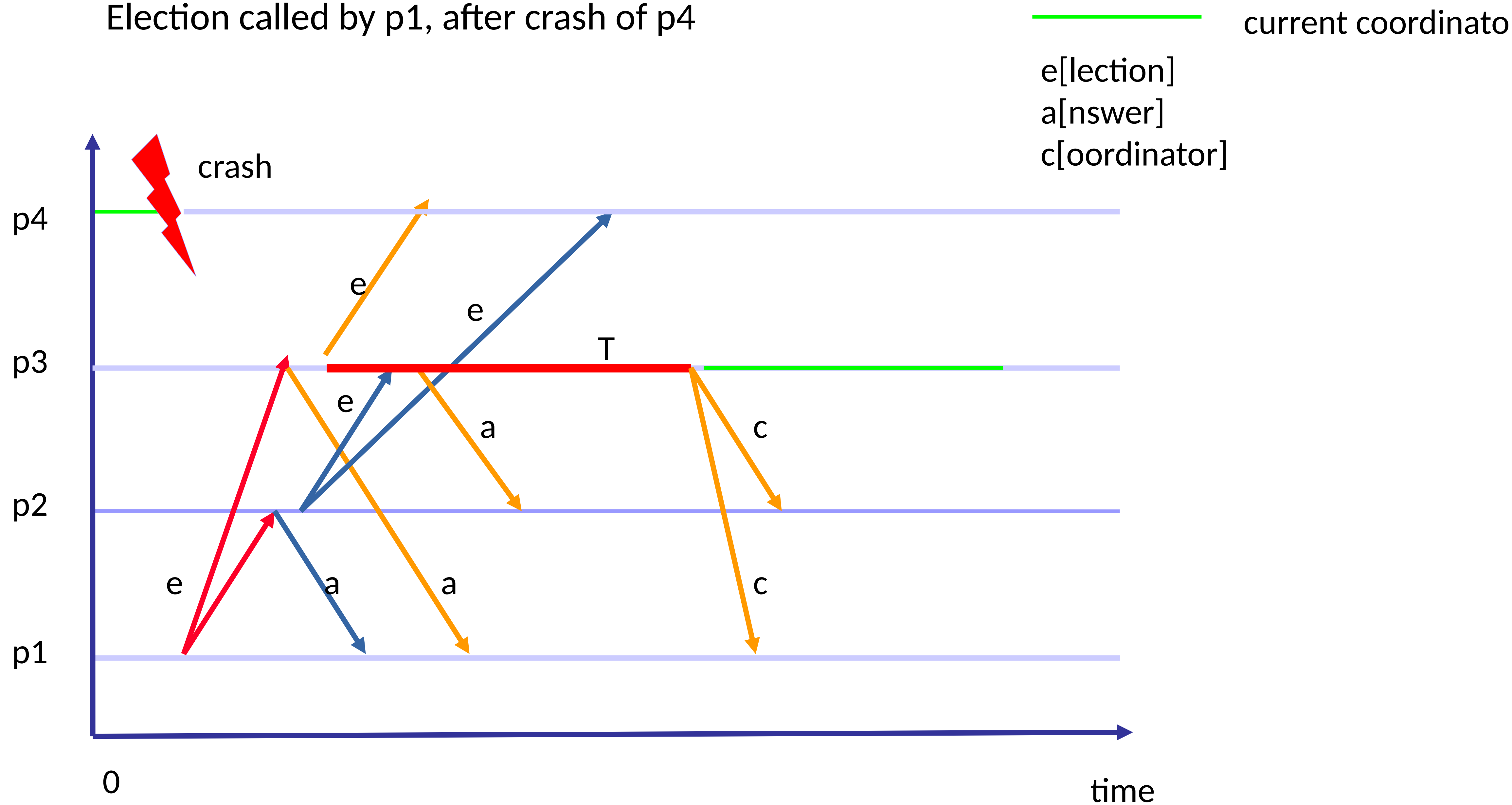
$elected_i$ = j

Receipt of election(j)-message at pi

if(no elections initiated by $\mathbf{p_i}$) {
send **answer-message** to $\mathbf{p_j}$
$\mathbf{p_i}$ calls election
}

Example

Election called by p1, after crash of p4



Algorithm OK ?

1

Safety

- Ok, if no process replacement
- If process replacement occurs & new process has highest ID
  - Duplicate coordinator messages
    - One from the replacing process
    - One from the largest-but-one ID

2

Liveness

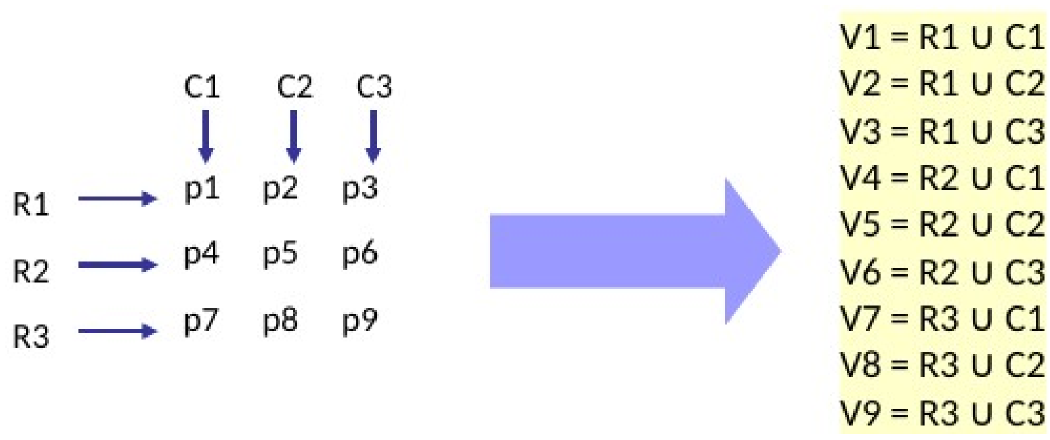
messages delivered reliably (no communication faults) either

- answer from **L**
- process is coordinator itself
- in any case coordinator identified !

**Distributed
Mutual Exclusion**

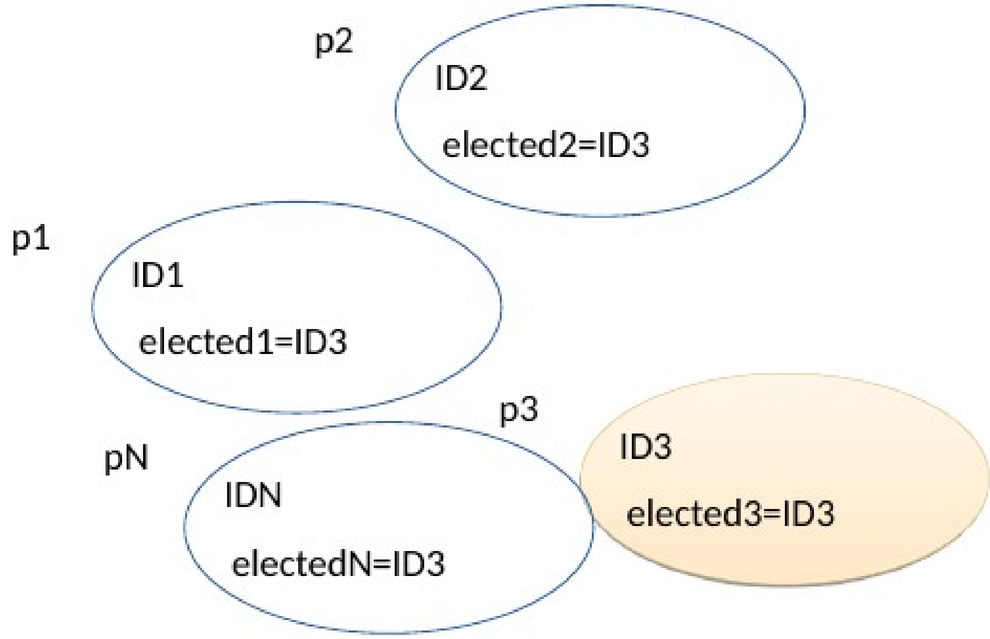
**Election
Mechanisms**

Summarizing



Coordination

- Maekawa Voting Algorithm
- Improvement over Ricard-Agrawala



Election

- Select a node for a special role
- Desirable properties & evaluation metrics
- Ring Algorithm
- Bully Algorithm

Lots of energy and success in your exams!

Questions?

Coordination

Part-2

Coordination