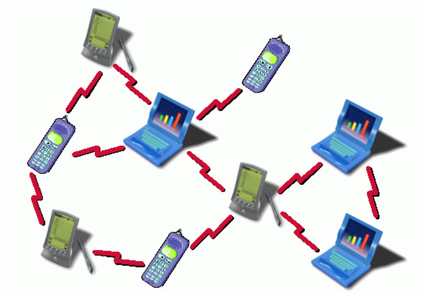
Chapter 5

Coordination

1. **Failure detection**
2. **Distributed mutual exclusion**
3. **Election**
4. **Ordered multicast**

General Problem

Given a set of processes $\Pi=\{p_i\}$, distributed over multiple hosts
- how to coordinate actions ?
- agree on contents of shared variables ("global state") ?

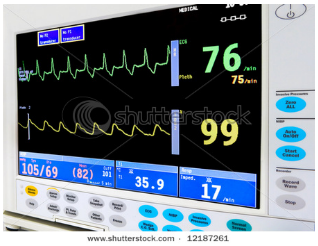


Example problems
- control access to common database (locking)
- elect central node in ad hoc network
- elect time server in network
- avoid static master-slave relations to enhance robustness

Chapter 6

Coordination

1. **Failure detection**
    1. Problem statement
    2. Algorithm
2. **Distributed mutual exclusion**
3. **Election**
4. **Ordered multicast**



Detecting failures

Process p still running ?

Terminology

Unreliable failure detector : only hints

possible states assigned to p

suspected : the process *p* is **probably** crashed

unsuspected : the process *p* is **probably** alive

Reliable failure detector : SURE about crashing

possible states

failed : the process *p* has **definitely** crashed

unsuspected : the process is **probably** still alive

Algorithms

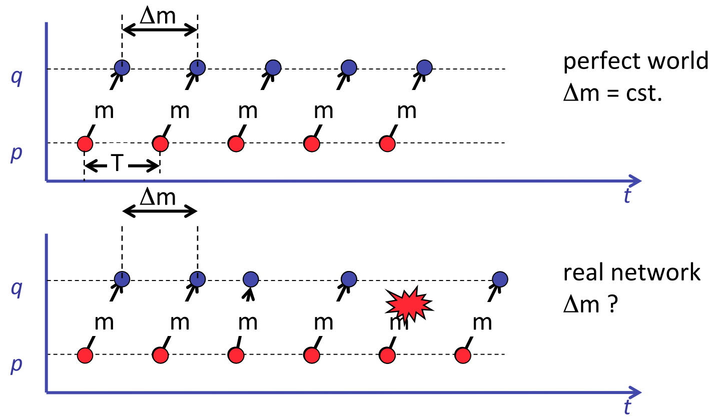
Two mechanisms

- heart beat from p
- server polls p

Heart beat

p sends message m to q, periodicity T

q measures inter arrival time of m : Δm



perfect world
Δm = cst.

real network
Δm ?

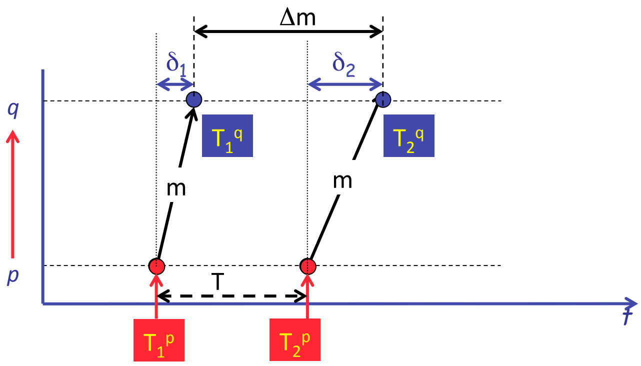
Algorithms

1. Failure detection
1. Problem statement

Synchronous system

- bounded network delay $\delta < \Delta$
- message is delivered

clock offset o
$C_q = C_p + o$



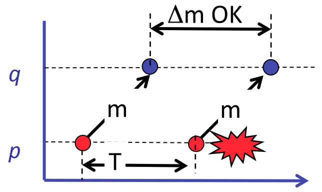
$$T_2^q = T_2^p + \delta_2 + o$$
$$T_1^q = T_1^p + \delta_1 + o$$

$$\Delta m = T_2^q - T_1^q = T_2^p - T_1^p + \delta_2 - \delta_1$$

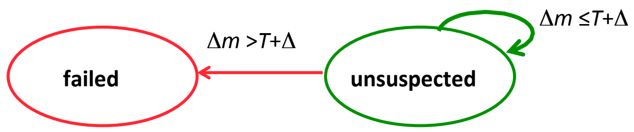
Synchronous system

$\Delta m = T_2^q - T_1^q$
$= T_2^p - T_1^p + \delta_2 - \delta_1$
$= T + \delta_2 - \delta_1$

$|\Delta m| = |T + \delta_2 - \delta_1|$
$\leq T + |\delta_2 - \delta_1|$
$\leq T + \Delta$



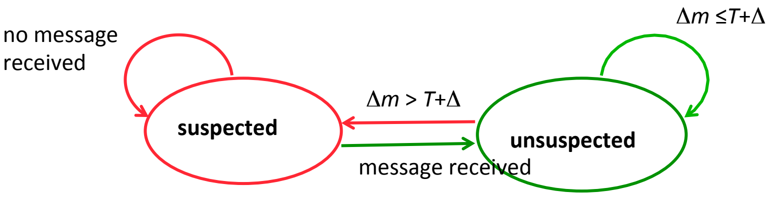
$\Delta m > T + \Delta$ -> p has crashed
$\Delta m \leq T + \Delta$ -> p WAS alive,
but MIGHT have crashed since



Reliable failure detector

Asynchronous system

No assumption on network delay
Only suspicion if Δm becomes exceedingly large
-> Limit Δ on Δm



Unreliable failure detector

Δ ?

Δ >> too many unsuspected
Δ << too many suspected
sensible choice : set Δ in relation to network delay
- adapt dynamically
- e.g. $\Delta = 1.2\ \delta$

Chapter 6

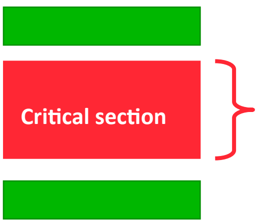
Coordination

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

Critical sections

Goal

Coordinate process access to shared resources



Accesses common resource
No other process should access same resource

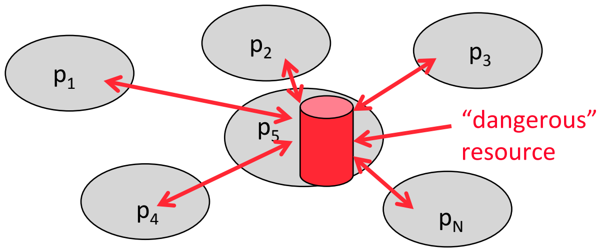
Distributed **mutual exclusion**

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

Problem statement

Consider
- **N processes {p_1, … , p_N}, NO shared variables**
- **access common resources in a critical section**
- **asynchronous system**
- **processes CAN communicate (know each other)**

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



A good solution should ...

1. safety [REQUIRED]
At most ONE process may execute in critical section at any time

2. liveness [REQUIRED]
Requests to enter/leave critical sections eventually succeed
-> deadlock-free algorithm
-> no starvation

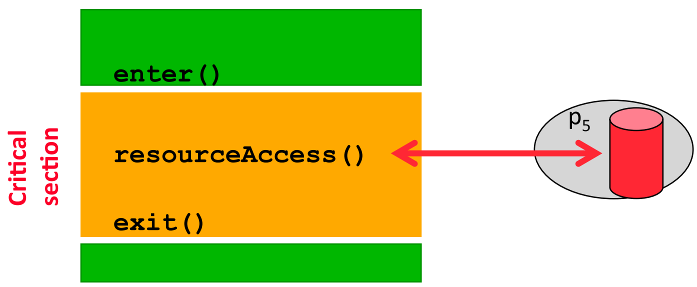
3. fairness [BONUS]
Access to critical section is granted using "happened - before" relation
-> use logical clock to order access requests

Critical section access API

2. Mutual Exclusion
   1. Problem statement

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



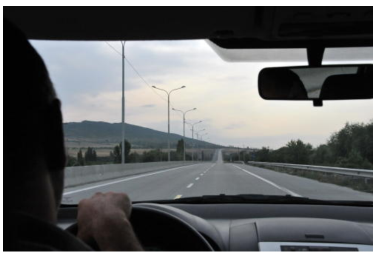
13

How to quantify solution quality ?

2. Mutual Exclusion
   2. Evaluation Metrics

Evaluation metrics

- **bandwidth consumption**
  = number of messages sent to enter/leave critical section
- **client delay**
  - = time needed to enter/leave critical section
  - measured in **UNLOADED** system

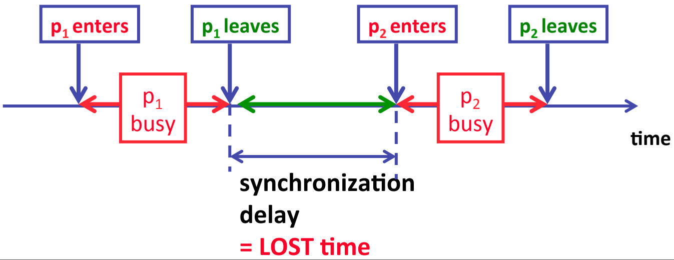


14

How to quantify solution quality ?

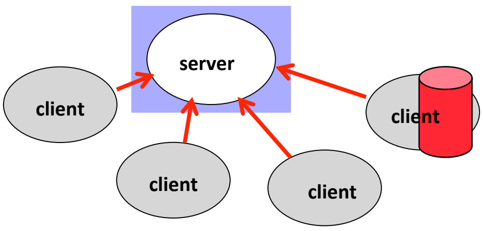
- system throughput
  - how many processes can access critical section in given time period ?
  - depends on **resourceAccess()**-time
  - derived measure: synchronization delay =
    average (time process (i+1) enters – time process (i) leaves)

LOADED system



Central server

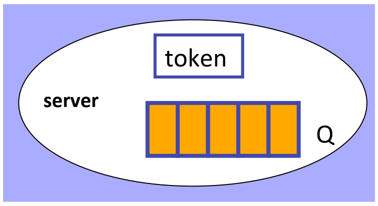
Centralized algorithms (one server)
- easy
- but typically poor scaling



Messages

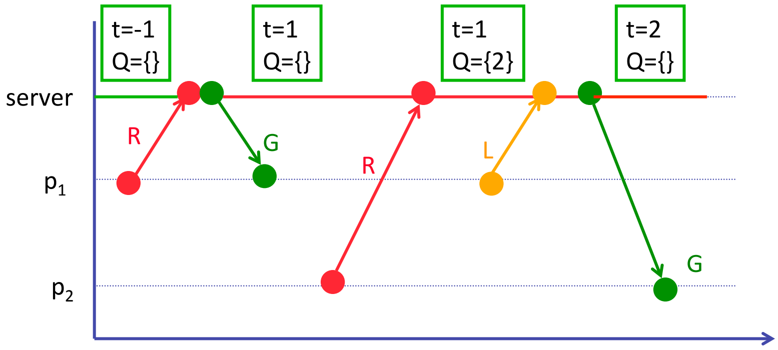
Client -> Server : **Request**
Client -> Server : **Leave**
Server -> Client : **Grant**

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

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)

server side

Algorithm OK ?

Safety

guarded by token variable

Liveness

Request to enter

all processes eventually leave
each leave dequeues a message from Q
if oldest Request dequeued
=> every Request eventually handled

Request to leave

no permission needed from server

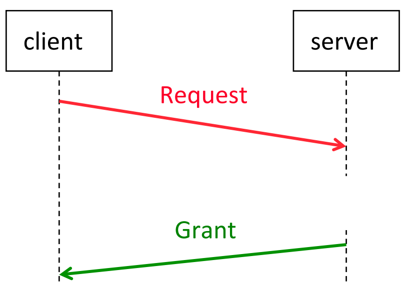
Fairness

Order Q according to "happened-before"

Algorithm efficient ?

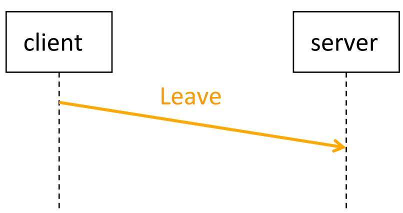
Bandwidth

enter()



=> 2 messages/enter

leave()



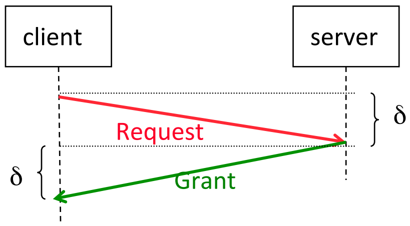
=> 1 message/leave

Algorithm efficient ?

Client delay (unloaded system)

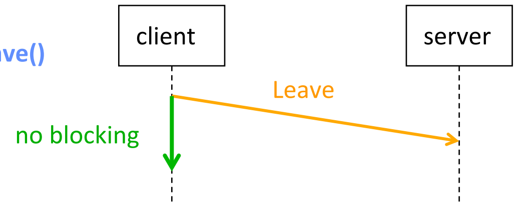
δ=time needed for 1 communication
RTT = 2δ

enter()



=> 2δ

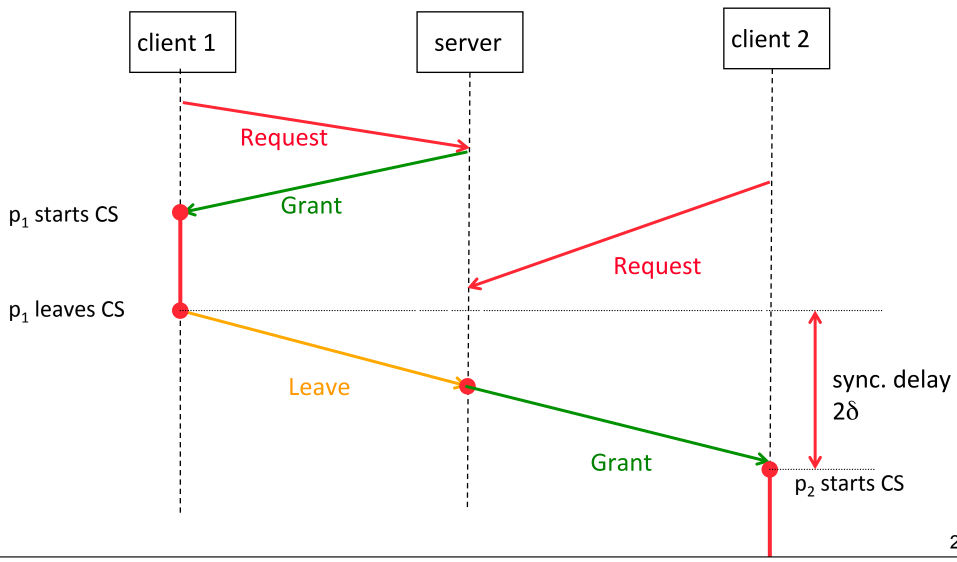
leave()



=> 0δ

Algorithm efficient ?

Synchronization delay (loaded system)

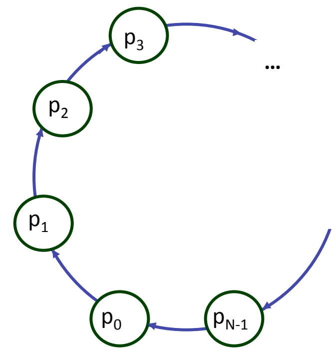


Ring-based algorithm

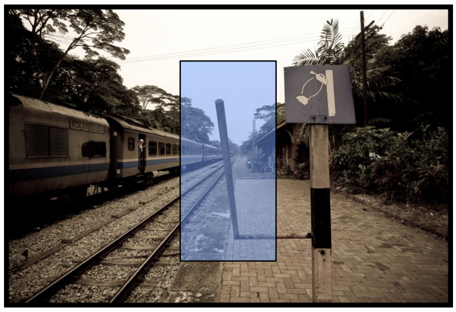
2. Mutual Exclusion
   4. Ring approach

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

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



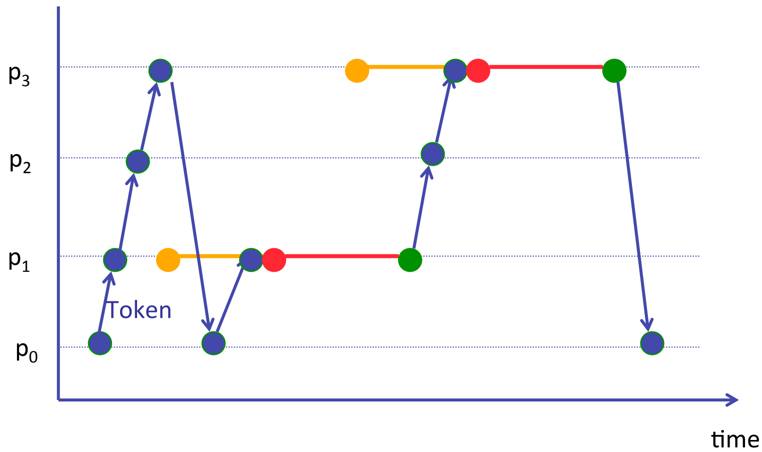
Message : **Token**



23

Ring-based algorithm

2. Mutual Exclusion
   4. Ring approach



- process wants critical section
- process executes critical section
- process leaves critical section

24

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

Safety

process can only send Token if it has received Token

Liveness

process eventually leave
=> Token circulates in the ring
(p not allowed new access !)
=> No starvation

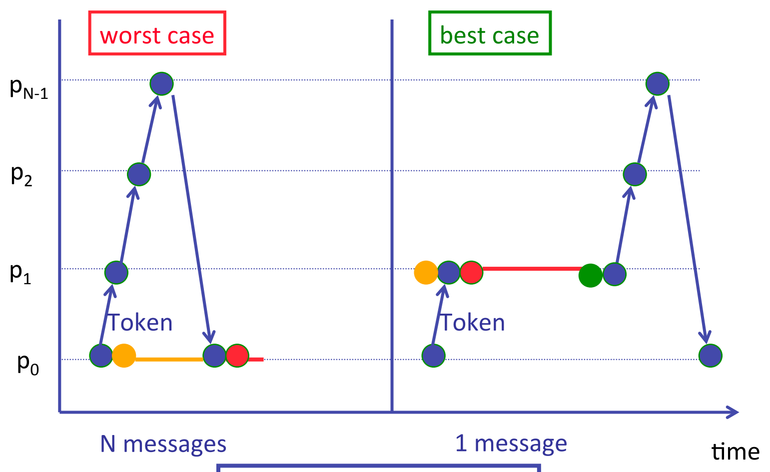
Fairness

not guaranteed : processes need luck for early access
access order not based "happened before" of requests

Algorithm efficient ?

Bandwidth

enter



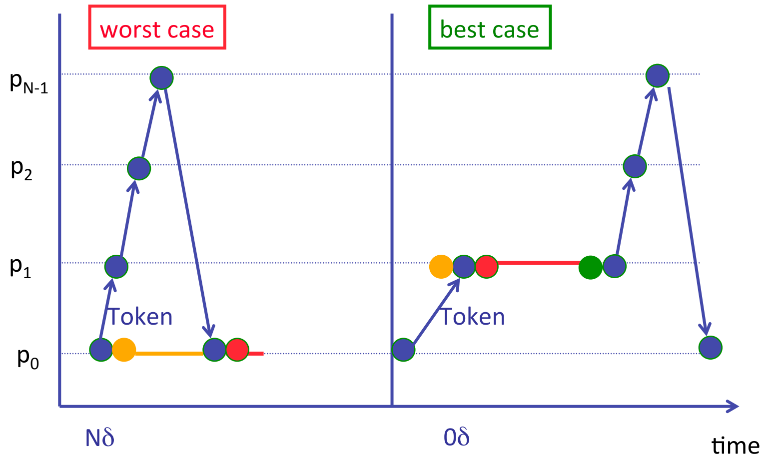
average : (N+1)/2 messages

leave : included in enter
BUT : algorithm always consumes bandwidth !

Algorithm efficient ?

Client delay (unloaded system)

enter()

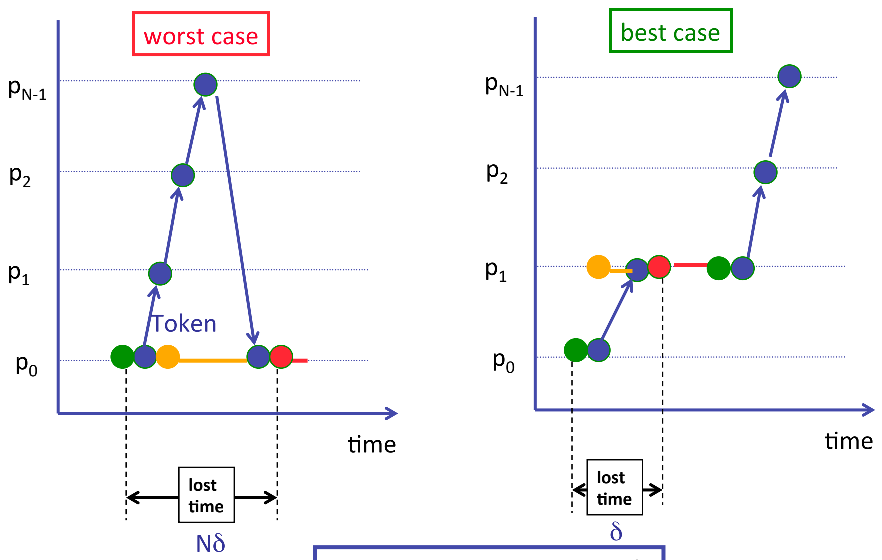


leave() : no blocking for process to leave critical section
0δ

average client delay : Nδ/2

Algorithm efficient ?

Synchronization delay (loaded system)

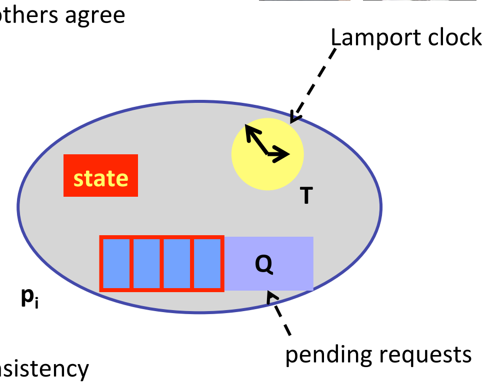


average sync. delay : (N+1)δ/2

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



Organization of Q ?
- Lamport-clock based happened-before
- Total ordering implemented to ensure consistency

$$<p_1,T_1> \; < \; <p_2,T_2>$$
$$\Leftrightarrow \quad (T_1<T_2) \text{ or } ((T_1==T_2) \text{ and } (p_1<p_2))$$

Ricart-Agrawala algorithm

Messages
Request(p_i,T_i),
Reply

each process p_i

Initialization
state = Released

enter()
state = Wanted
multicast Request(p_i,T_i) to all processes
store T=T_i 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(p_j,T_j) at p_i (i≠j)
```
if (state == Held) or ((state==Wanted) and (T,p_i)<(T_j,p_j)) {
    enqueue Request in Q
    // do NOT reply
} else send Reply to p_j
```

Examples

(1) Process p requests entry, all others in RELEASED-state

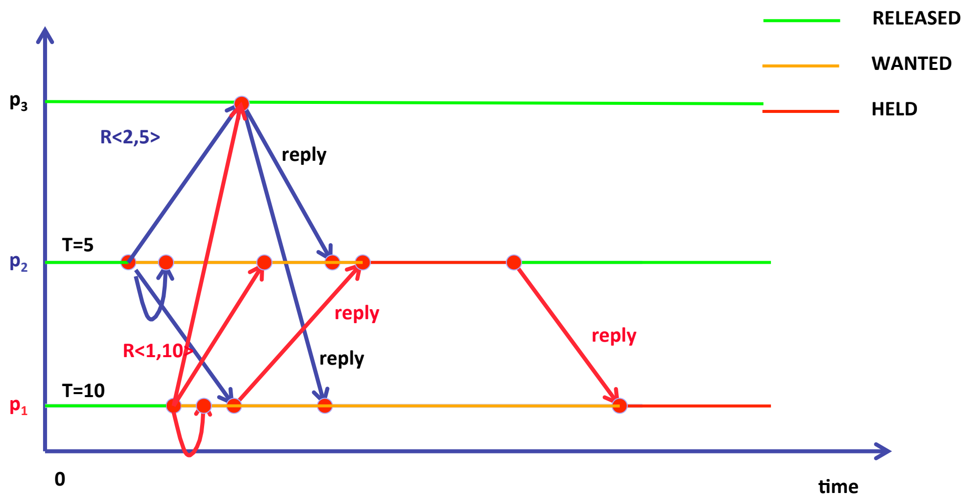
-> all (N-1) other processes reply immediately
-> p can enter CS

(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 p_1 and p_2 request entry, p_3 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 $(T,p_i)<(T_j,p_j)$)
DOES NOT hold for
$(p_j=q),(p_i=p)$ (a)
AND $(p_j=p),(p_i=q)$ (b)

some logic for (a)
=> ![($state_p$ == Held) or (($state_p$==Wanted) and $(T_p,p)<(T_q,q)$)]
=> ($state_p$!=Held) AND [($state_p$!=Wanted) or $(T_q,q)<(T_p,p)$]
=> [($state_p$!= Held) and ($state_p$!=Wanted)]
or [($state_p$!= Held) and $(T_q,q)<(T_p,p)$]
=> ($state_p$==Released) or [($state_p$!= Held) and $(T_q,q)<(T_p,p)$]

p can NOT be in state Released (because now in CS)
=> [($state_p$!= Held) and $(T_q,q)<(T_p,p)$]

Algorithm OK ?

(1) Safety

...
(a) => [($state_p$!= Held) and $(T_p,p)>(T_q,q)$]
(b) => [($state_q$!= Held) and $(T_q,q)>(T_p,p)$]

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

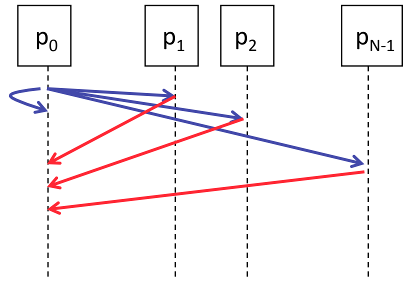
(3) Fairness

Requests replied in happened-before order

Algorithm efficient ?

(1) Bandwidth

enter()



N Request messages
(N-1) Reply messages

if multicast supported
1 Request message only !

leave() **no messages needed for leaving**

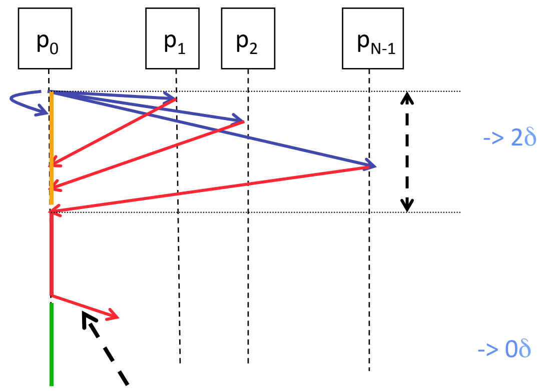
35

Algorithm efficient ?

(2) Client delay (unloaded system)

enter()

leave()

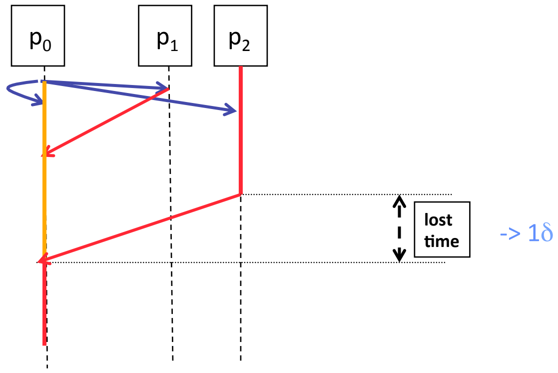


Reply to pending Request
does not block p_0

36

Algorithm efficient ?

(3) Synchronisation delay (loaded system)



Maekawa Voting

Filosophy

- Drawback of Ricart-Agrawala : all processes need to agree
- 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 p_i : V_i

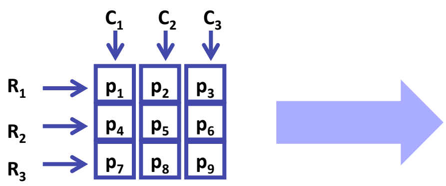
- **$V_i \subseteq \{p_1, \ldots, p_N\}$, satisfying (Required)**
- **$p_i \in V_i$ (Required)**
- **$V_i \cap V_j \neq \varnothing$ (Required)**
- **$|V_i|$ = K (fairness) (Bonus)**
- **p_j contained in M voting sets (Bonus)**

Algorithm

process $q \in (V_i \cap V_j)$
q votes for 1 process only
-> q makes sure p_i and p_j are not
simulataneously executing critical section
-> safety condition met
Additional state needed per process : **voted**

Constructing voting sets

- **Choosing parameters**
  - Theoretical result : optimal solution (minimal K)
    - $K \approx N^{1/2}$
    - $M = K$
  - In practice : difficult to calculate optimal V_i
    Sub-optimal solution
    - $K \approx 2N^{1/2}$
    - $M = K$
- **Practical algorithm** (for $N=S^2$)
  for S=3



$V_1 = R_1 \cup C_1$
$V_2 = R_1 \cup C_2$
$V_3 = R_1 \cup C_3$
$V_4 = R_2 \cup C_1$
$V_5 = R_2 \cup C_2$
$V_6 = R_2 \cup C_3$
$V_7 = R_3 \cup C_1$
$V_8 = R_3 \cup C_2$
$V_9 = R_3 \cup C_3$

Constructing voting sets

In general

Construct SxS matrix **A**, consisting of all processes
- i is row where p is found
- j is column where q is found
- R_i = i-th row of **A**
- C_j = j-th column of **A**

Voting set $V_p = R_i \cup C_j$

Checking this V_p ...

$p \in R_i$, $p \in C_j$ => $p \in V_p$

$V_p=R_i \cup C_j$
$V_q=R_s \cup C_t$
=> $V_p \cap V_q = (R_i \cup C_j) \cap (R_s \cup C_t)$
$= (R_i \cap R_s) \cup (R_i \cap C_t) \cup (C_j \cap R_s) \cup (C_j \cap C_t)$

BUT $R_k \cap C_l \neq F$ (for any k,l) ($s_{kl} \in R_k$ and $s_{kl} \in C_l$)
=> $V_p \cap V_q \neq F$

K = 2S -1
K = M

The algorithm

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)

Messages

Request
Reply
Release

The algorithm

Initialization

state = Released
voted = False

enter()

state = Wanted
multicast Request(p_i) to voting set V_i
wait until K Reply messages received
state = Held

leave()

state = Released
multicast Release to voting set V_i

when receiving Request(p_j) at p_i

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

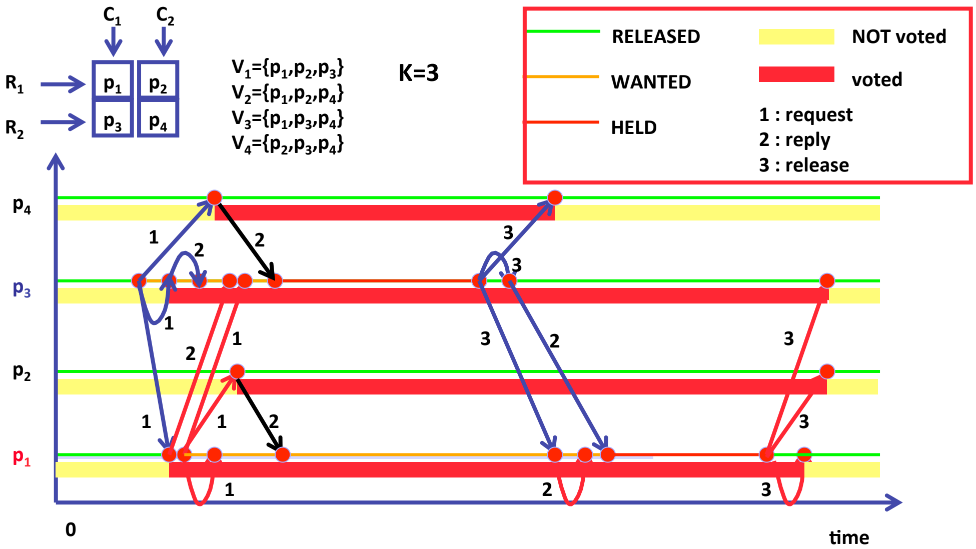
when receiving Release(p_j) at p_i

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

each process p

Example

Processes p_1 and p_3 request entry, p_2 and p_4 not



Algorithm OK ?

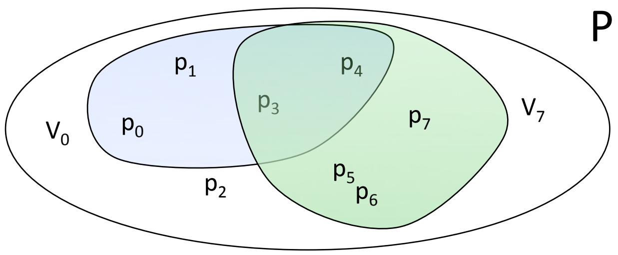
(1) Safety

Suppose p and q simultaneously active
=> all processes in V_p and V_q voted for p AND q
=> process t in $V_p \cap V_q \neq F$ voted for p AND q
impossible :

voted$_t$=True immediately after voting for 1 process

(2) Liveness

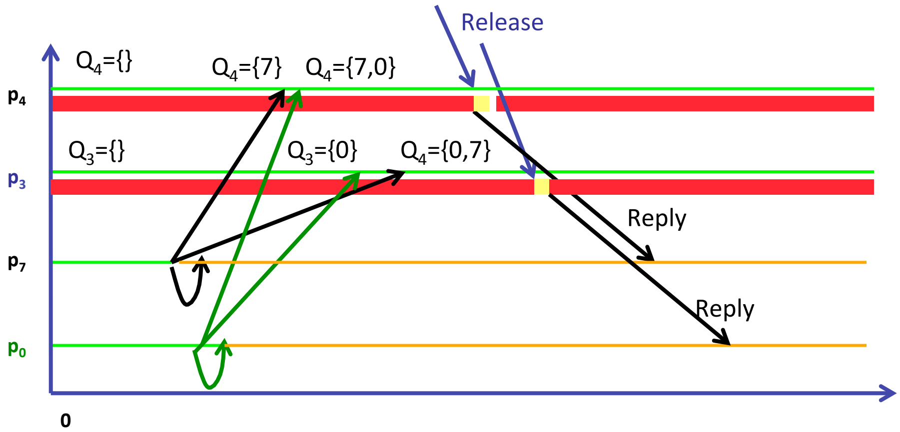
deadlock prone !!!



$V_0 \cap V_7 = \{p_3,p_4\}$

should vote consistently !!!

Algorithm NOT OK !



p_7 and p_0 will **NEVER** receive Reply from **complete** voting set !

Algorithm OK ?

(2) Liveness

Solution : organise Q according to Lamport clock !

(3) Fairness

OK by adaptation !

Algorithm efficient ?

(1) Bandwidth

enter()

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

-> K Request messages
-> K Reply messages

leave()

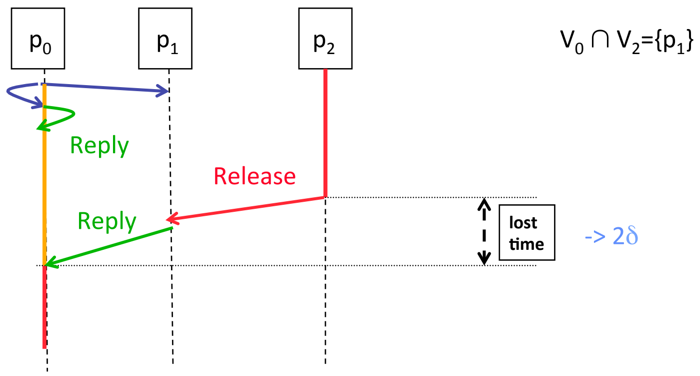
explicit Leave message now needed
-> K Release messages

(2) Client delay

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

Algorithm efficient ?

(3) Synchronization delay



Summary on efficiency

2. Mutual Exclusion
Summary

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

49

Chapter 6

Coordination

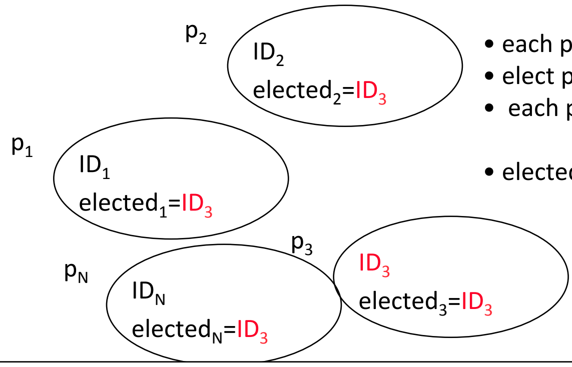
1. **Failure detection**
2. **Distributed mutual exclusion**
3. **Election**
    1. Problem statement
    2. Evaluation metrics
    3. Ring algorithm
    4. Multicast algorithm
4. **Ordered multicast**

50

The problem with elections

Consider

- N processes $\{p_1, \ldots, p_N\}$
  - NO shared variables
  - knowing each other (can communicate)
- select ONE process to play special role (e.g. coordinator)
- every process p_i should have same coordinator
- if elected process fails : do new election round



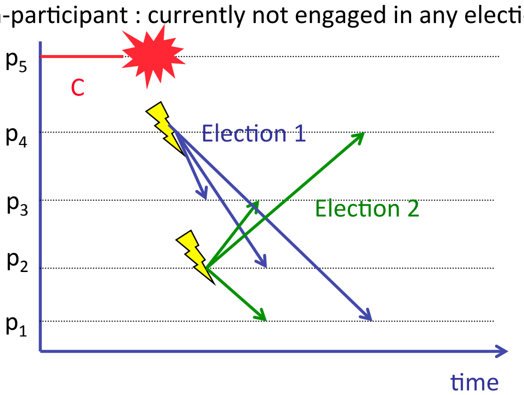
- each process has unique ID
- elect process with largest ID
- each process has state
  $elected_i$ = ID of elected process
- $elected_i$ should be independent of i

Coordinator

Some terms

Terminology

- 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 p_i has :

$elected_i$ = ?

OR $elected_i$ = P

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

(2) liveness (REQUIRED)

- all processes p_i participate
- eventually set $elected_i \neq$? or crash

Evaluation metrics

- **bandwidth**

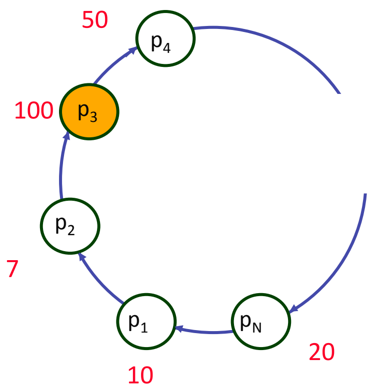
  \# messages needed to do election process

- **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 model

No failures :

- reliable channels
- no process crashes

Asynchronous system

Ring algorithm: Chang – Roberts

Messages

Election(i,ID)
- i = initiator of election
- ID = current max ID

Elected(i)
- process p_i has been elected

Each process has

state : participant$_i$ (True/False)
elected$_i$ (ID of elected process, or ?)

Algorithm

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

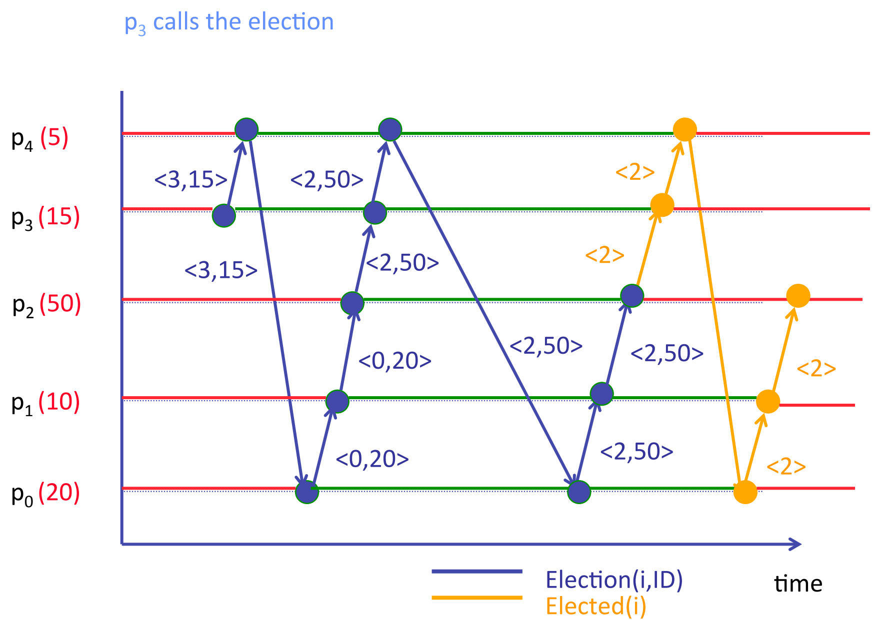
```
if(i≠j) {
        participant_j = FALSE
        elected_j = i
        forward Elected(i)
}
```

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

```
if(ID>ID_j) {
        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)
}
```

each process p

Example



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 **IDs are unique**
(ID_p<ID_q) => q will NOT forward Elected(p,ID_p)
(ID_p>ID_q) => p will NOT forward Elected(q,ID_q)
=> 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)

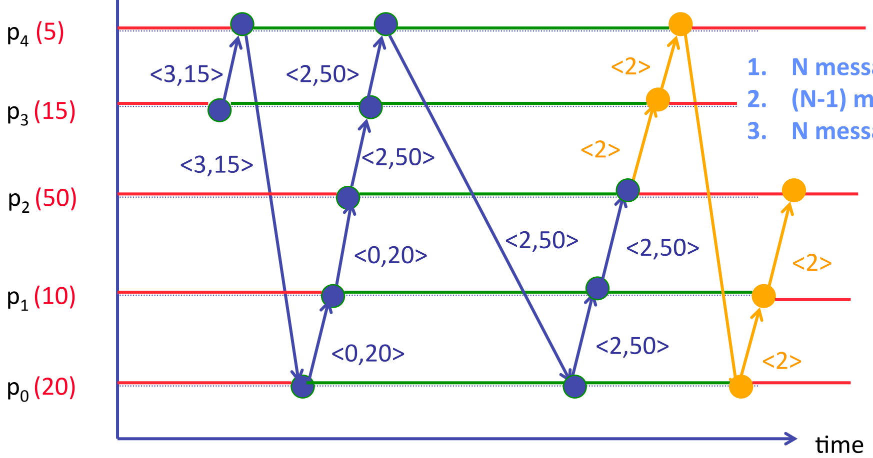
Algorithm efficient ?

3. Elections
3. Ring algorithm

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

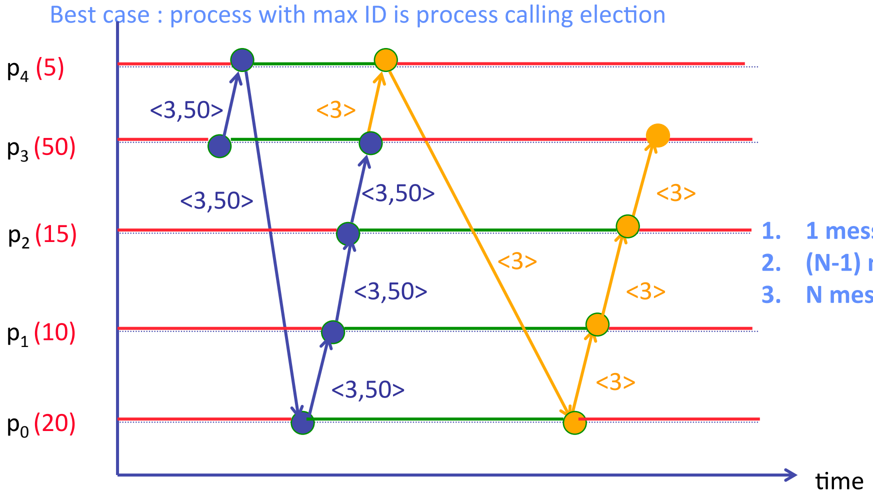


1. **N messages**
2. **(N-1) messages**
3. **N messages**

Algorithm efficient ?

3. Elections
3. Ring algorithm

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 model

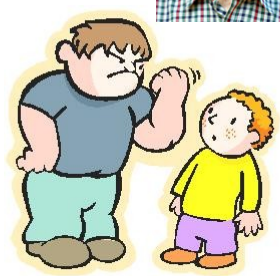
process crashes dealt with

System model

Synchronous system (uses time-outs to detect failure)

A-priori knowledge

process knows all processes with larger ID



Philosophy

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)

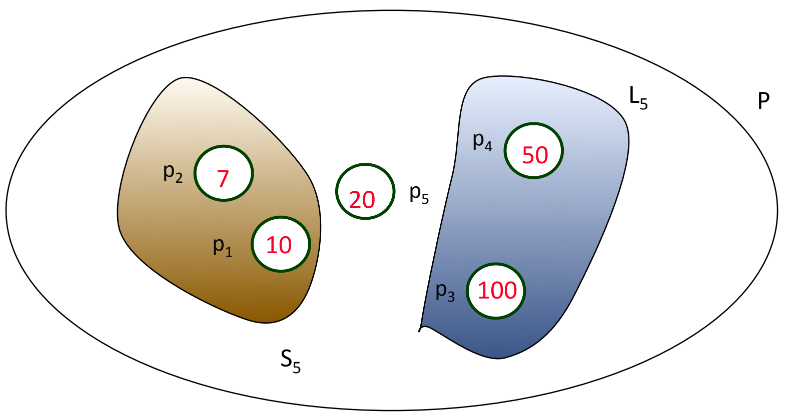
set *S* of other processes (smaller IDs)

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 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 p_i

$elected_i = j$

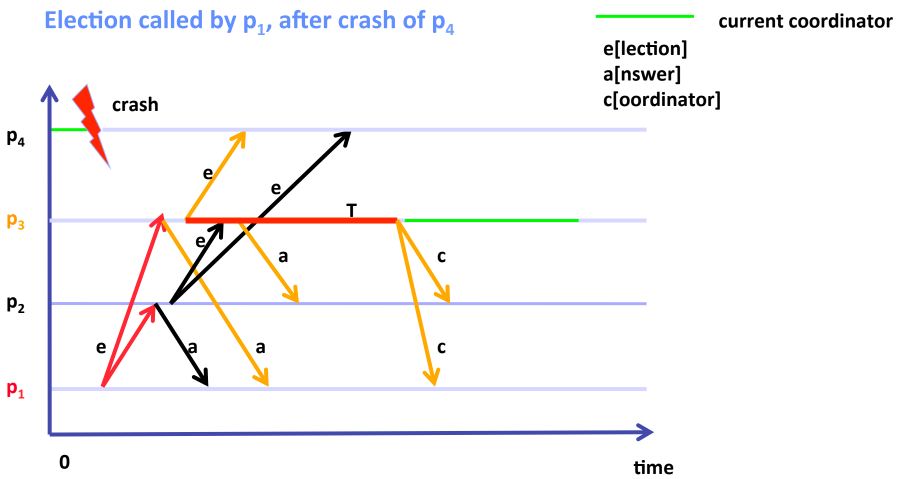
Receipt of election(j)-message at p_i

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

If new process started with highest ID
-> will become coordinator (bully !)

Example

Election called by p_1, after crash of p_4



Algorithm OK ?

safety

OK IF NO PROCESS REPLACEMENT

IF PROCESS REPLACEMENT OCCURS
If new process has highest ID
-> announces coordinator
-> can conflict with other announcement (if election running)

liveness

messages delivered reliably (no communication faults)
either
- answer from L
- process is coordinator itself

-> in any case coordinator identified !

Chapter 6

Coordination

1. Failure detection
2. Distributed mutual exclusion
3. Election
4. **Ordered multicast**

Multicast

Processes can be part of multiple multicast-groups

Basic operations

multicast(g,m)	send message m to all members of group g
deliver(m)	deliver received message m to the application

Each message carries

sender(m)
group(m)
payload(m)

closed group : members only
open group : process not part of group g can multicast to g

Basic-Multicast

B-multicast over reliable message delivery

send(p,m)	reliable send m to p
receive(m)	put message in input queue of p

B-multicast(g,m) : for each process p in g, send(p,m)

On receive(m) at p : B-deliver(m) at p

Issues

- problem of ack-implosion (for large number of processes)
- inefficient usage of network bandwidth (causing delay)

Extentions to B-multicast

- provide reliability (guaranteed delivery, exactly once)
- implementation over IP-multicast
- guarantee message ordering

Ordering

FIFO-ordering

if a correct process issues

multicast(g,m)

multicast(g,m')

then every correct process delivering m' will deliver m before m'

Causal ordering

if multicast(g,m) "happened before" multicast(g,m'),

every correct process that delivers m' will deliver m before m'

Total ordering

if a correct process delivers m before delivering m',

then every correct process that delivers m' will deliver m before m'

Reliability NOT implied

Total ordering does NOT imply FIFO- or causal ordering

-> FIFO-total ordering

-> causal-total ordering